Working Draft, Standard for Programming Language C++

Note: this is an early draft. It’s known to be incomplet 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 7
  4.1 Implementation compliance 7
  4.2 Structure of this document 8
  4.3 Syntax notation 8
  4.4 Acknowledgments 9
5 Lexical conventions 10
  5.1 Separate translation 10
  5.2 Phases of translation 10
  5.3 Character sets 11
  5.4 Preprocessing tokens 12
  5.5 Alternative tokens 13
  5.6 Tokens 13
  5.7 Comments 13
  5.8 Header names 14
  5.9 Preprocessing numbers 14
  5.10 Identifiers 14
  5.11 Keywords 15
  5.12 Operators and punctuators 16
  5.13 Literals 16
6 Basics 26
  6.1 Preamble 26
  6.2 Declarations and definitions 26
  6.3 One-definition rule 28
  6.4 Scope 32
  6.5 Name lookup 38
  6.6 Program and linkage 51
  6.7 Memory and objects 54
  6.8 Types 68
  6.9 Program execution 74
7 Expressions 86
  7.1 Preamble 86
  7.2 Properties of expressions 87
  7.3 Standard conversions 89
  7.4 Usual arithmetic conversions 94
  7.5 Primary expressions 95
  7.6 Compound expressions 110
  7.7 Constant expressions 140
8 Statements 146
  8.1 Preamble 146
  8.2 Labeled statement 147
  8.3 Expression statement 147
  8.4 Compound statement or block 147
  8.5 Selection statements 147
  8.6 Iteration statements 149
13.8 Name resolution .......................................................... 385
13.9 Template instantiation and specialization ............................... 401
13.10 Function template specializations ...................................... 413

14 Exception handling .......................................................... 433
  14.1 Preamble ................................................................. 433
  14.2 Throwing an exception .................................................. 434
  14.3 Constructors and destructors ......................................... 435
  14.4 Handling an exception .................................................. 436
  14.5 Exception specifications ................................................. 437
  14.6 Special functions ....................................................... 440

15 Preprocessing directives ..................................................... 442
  15.1 Preamble ................................................................. 442
  15.2 Conditional inclusion .................................................... 444
  15.3 Source file inclusion ..................................................... 446
  15.4 Module directive ........................................................ 447
  15.5 Header unit importation ................................................ 447
  15.6 Macro replacement ....................................................... 449
  15.7 Line control ............................................................. 454
  15.8 Error directive .......................................................... 454
  15.9Pragma directive ........................................................ 454
  15.10 Null directive .......................................................... 455
  15.11 Predefined macro names ............................................... 455
  15.12Pragma operator ........................................................ 457

16 Library introduction .......................................................... 458
  16.1 General ................................................................. 458
  16.2 The C standard library .................................................. 459
  16.3 Definitions .............................................................. 459
  16.4 Method of description ................................................... 462
  16.5 Library-wide requirements ............................................. 468

17 Language support library ..................................................... 488
  17.1 General ................................................................. 488
  17.2 Common definitions ..................................................... 488
  17.3 Implementation properties ............................................. 492
  17.4 Integer types ........................................................... 502
  17.5 Startup and termination ............................................... 503
  17.6 Dynamic memory management ....................................... 504
  17.7 Type identification .................................................... 511
  17.8 Source location ......................................................... 513
  17.9 Exception handling .................................................... 515
  17.10 Initializer lists ........................................................ 519
  17.11 Comparisons ........................................................... 520
  17.12 Coroutines ............................................................ 527
  17.13 Other runtime support ................................................ 532

18 Concepts library ............................................................ 534
  18.1 General ................................................................. 534
  18.2 Equality preservation ................................................... 534
  18.3 Header <concepts> synopsis ......................................... 535
  18.4 Language-related concepts ............................................. 537
  18.5 Comparison concepts ................................................... 542
  18.6 Object concepts ........................................................ 545
  18.7 Callable concepts ....................................................... 545
## 19 Diagnostics library
- 19.1 General ........................................ 547
- 19.2 Exception classes ............................. 547
- 19.3 Assertions ..................................... 550
- 19.4 Error numbers ................................. 550
- 19.5 System error support ......................... 552

## 20 General utilities library
- 20.1 General ........................................ 561
- 20.2 Utility components ............................ 561
- 20.3 Compile-time integer sequences .............. 566
- 20.4 Pairs .......................................... 567
- 20.5 Tuples ......................................... 571
- 20.6 Optional objects ............................... 581
- 20.7 Variants ....................................... 593
- 20.8 Storage for any type ......................... 604
- 20.9 Bitsets ................................-------- 609
- 20.10 Memory ........................................ 614
- 20.11 Smart pointers ............................... 630
- 20.12 Memory resources ............................. 653
- 20.13 Class template scoped_allocator_adaptor .. 662
- 20.14 Function objects .............................. 666
- 20.15 Metaprogramming and type traits .......... 689
- 20.16 Compile-time rational arithmetic .......... 713
- 20.17 Class type_index ............................. 715
- 20.18 Execution policies ........................... 717
- 20.19 Primitive numeric conversions ............. 718
- 20.20 Formatting .................................. 721

## 21 Strings library
- 21.1 General ........................................ 739
- 21.2 Character traits .............................. 739
- 21.3 String classes ................................. 744
- 21.4 String view classes ......................... 770
- 21.5 Null-terminated sequence utilities ......... 779

## 22 Containers library
- 22.1 General ........................................ 785
- 22.2 Container requirements ...................... 785
- 22.3 Sequence containers ......................... 818
- 22.4 Associative containers ....................... 846
- 22.5 Unordered associative containers .......... 864
- 22.6 Container adaptors ........................... 886
- 22.7 Views ......................................... 894

## 23 Iterators library
- 23.1 General ........................................ 901
- 23.2 Header <iterator> synopsis ................. 901
- 23.3 Iterator requirements ....................... 908
- 23.4 Iterator primitives ........................... 928
- 23.5 Iterator adaptors ............................. 931
- 23.6 Stream iterators .............................. 952
- 23.7 Range access ................................ 957

## 24 Ranges library
- 24.1 General ........................................ 960
- 24.2 Header <ranges> synopsis ................... 960
- 24.3 Range access ................................ 965
- 24.4 Range requirements ......................... 969
| 24.5 | Range utilities | 972 |
| 24.6 | Range factories | 977 |
| 24.7 | Range adaptors | 987 |

### 25 Algorithms library
- 25.1 General
- 25.2 Algorithms requirements
- 25.3 Parallel algorithms
- 25.4 Header `<algorithm>` synopsis
- 25.5 Algorithm result types
- 25.6 Non-modifying sequence operations
- 25.7 Mutating sequence operations
- 25.8 Sorting and related operations
- 25.9 Header `<numeric>` synopsis
- 25.10 Generalized numeric operations
- 25.11 Specialized `<memory>` algorithms
- 25.12 C library algorithms

### 26 Numerics library
- 26.1 General
- 26.2 Numeric type requirements
- 26.3 The floating-point environment
- 26.4 Complex numbers
- 26.5 Bit manipulation
- 26.6 Random number generation
- 26.7 Numeric arrays
- 26.8 Mathematical functions for floating-point types
- 26.9 Numbers

### 27 Time library
- 27.1 General
- 27.2 Header `<chrono>` synopsis
- 27.3 Cpp17Clock requirements
- 27.4 Time-related traits
- 27.5 Class template `duration`
- 27.6 Class template `time_point`
- 27.7 Clocks
- 27.8 The civil calendar
- 27.9 Class template `hh_mm_ss`
- 27.10 12/24 hours functions
- 27.11 Time zones
- 27.12 Formatting
- 27.13 Parsing
- 27.14 Header `<ctime>` synopsis

### 28 Localization library
- 28.1 General
- 28.2 Header `<locale>` synopsis
- 28.3 Locales
- 28.4 Standard `<locale>` categories
- 28.5 C library locales

### 29 Input/output library
- 29.1 General
- 29.2 Iostreams requirements
- 29.3 Forward declarations
- 29.4 Standard iostream objects
- 29.5 Iostreams base classes
- 29.6 Stream buffers
30 Regular expressions library

30.1 General ......................................................... 1486
30.2 Definitions ................................................. 1486
30.3 Requirements ............................................... 1487
30.4 Header <regex> synopsis ................................. 1488
30.5 Namespace std::regex_constants ......................... 1492
30.6 Class regex_error ........................................... 1495
30.7 Class template regex_traits ............................. 1495
30.8 Class template basic_regex ............................. 1497
30.9 Class template sub_match ............................... 1501
30.10 Class template match_results ......................... 1503
30.11 Regular expression algorithms ....................... 1508
30.12 Regular expression iterators ......................... 1512
30.13 Modified ECMAScript regular expression grammar .... 1517

31 Atomic operations library

31.1 General ......................................................... 1520
31.2 Header <atomic> synopsis ................................. 1520
31.3 Type aliases ............................................... 1524
31.4 Order and consistency .................................... 1524
31.5 Lock-free property ........................................ 1526
31.6 Waiting and notifying ..................................... 1526
31.7 Class template atomic_ref ............................. 1527
31.8 Class template atomic ................................. 1533
31.9 Non-member functions ................................... 1548
31.10 Flag type and operations .............................. 1548
31.11 Fences ...................................................... 1550

32 Thread support library

32.1 General ......................................................... 1551
32.2 Requirements ............................................... 1551
32.3 Stop tokens ................................................. 1553
32.4 Threads ....................................................... 1558
32.5 Mutual exclusion .......................................... 1565
32.6 Condition variables ...................................... 1583
32.7 Semaphore ................................................. 1590
32.8 Coordination types ...................................... 1592
32.9 Futures ...................................................... 1595

A Grammar summary

A.1 Keywords ..................................................... 1610
A.2 Lexical conventions ....................................... 1610
A.3 Basics ......................................................... 1614
A.4 Expressions ................................................ 1614
A.5 Statements .................................................. 1618
A.6 Declarations ............................................... 1619
A.7 Modules ...................................................... 1625
A.8 Classes ....................................................... 1626
A.9 Overloading .................................................. 1627
A.10 Templates ................................................... 1627
A.11 Exception handling ...................................... 1629
A.12 Preprocessing directives .............................. 1629
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

1 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.


(1.2) — ISO/IEC 2382 (all parts), _Information technology — Vocabulary_

(1.3) — ISO 8601:2004, _Data elements and interchange formats — Information interchange — Representation of dates and times_

(1.4) — ISO/IEC 9899:2018, _Programming languages — C_

(1.5) — ISO/IEC 9945:2003, _Information Technology — Portable Operating System Interface (POSIX)_

(1.6) — ISO/IEC 10646, _Information technology — Universal Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane_

(1.7) — ISO/IEC 10646-1:1993, _Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane_

(1.8) — ISO/IEC/IEEE 60559:2011, _Information technology — Microprocessor Systems — Floating-Point arithmetic_

(1.9) — 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 Clause 7 of ISO/IEC 9899:2018 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: References to ISO/IEC 10646-1:1993 are used only to support deprecated features (D.20). — end note]

1) With the qualifications noted in Clause 17 through Clause 32 and in C.6, the C standard library is a subset of the C++ standard library.
3 Terms and definitions [intro.defs]

1 For the purposes of this document, the terms and definitions given in ISO/IEC 2382-1:1993, the terms, definitions, and symbols given in ISO 80000-2:2009, and the following apply.

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

[(2.1)] ISO Online browsing platform: available at https://www.iso.org/obp

[(2.2)] IEC Electropedia: available at http://www.electropedia.org/

3 16.3 defines additional terms that are used only in Clause 16 through Clause 32 and Annex D.

4 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 [defns.access] access
<execution-time action> read (7.3.1) or modify (7.6.19, 7.6.1.5, 7.6.2.2) the value of an object

[Note 1 to entry: Only objects of scalar type can be accessed. Attempts to read or modify an object of class type typically invoke a constructor (11.4.4) or assignment operator (11.4.5); such invocations do not themselves constitute accesses, although they may involve accesses of scalar subobjects. — end note]

3.2 [defns.argument] argument
(function call expression) expression in the comma-separated list bounded by the parentheses (7.6.1.2)

3.3 [defns.argument.macro] argument
(function-like macro) sequence of preprocessing tokens in the comma-separated list bounded by the parentheses (15.6)

3.4 [defns.argument.throw] argument
(throw expression) operand of throw (7.6.18)

3.5 [defns.argumenttempl] argument
(template instantiation) constant-expression, type-id, or id-expression in the comma-separated list bounded by the angle brackets (13.4)

3.6 [defns.block] 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

3.7 [defns.block.stmt] block
(statement) compound statement (8.4)

3.8 [defns.cond.supp] 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]

3.9 [defns.diagnostic] diagnostic message
message belonging to an implementation-defined subset of the implementation’s output messages
3.10  
.dynamic type
(type of the most derived object (6.7.2) to which the glvalue refers

[Example: If a pointer (9.3.3.1) \(p\) 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 (9.3.3.2) are treated similarly.
—end example]

3.11  
.dynamic type
(type of the prvalue expression

3.12  
ill-formed program
(program that is not well-formed (3.32)

3.13  
implementation-defined behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

3.14  
implementation limits
restrictions imposed upon programs by the implementation

3.15  
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

3.16  
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]

3.17  
parameter
(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

3.18  
parameter
(identifier from the comma-separated list bounded by the parentheses immediately following the macro name

3.19  
parameter
(member of a template-parameter-list

3.20  
signature
(name, parameter-type-list (9.3.3.5), and enclosing namespace (if any)

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

3.21  
signature
(name with trailing requires-clause name, parameter-type-list (9.3.3.5), enclosing class, and trailing requires-clause (9.3)
3.22 [defns.signature.templ]
signature
\(\text{(function template) name, parameter-type-list (9.3.3.5), enclosing namespace (if any), return type, template-head, and trailing requires-clause (9.3) (if any)}\)

3.23 [defns.signature.templ.friend]
signature
\(\text{(friend function template with constraint involving enclosing template parameters) name, parameter-type-list (9.3.3.5), return type, enclosing class, template-head, and trailing requires-clause (9.3) (if any)}\)

3.24 [defns.signature.spec]
signature
\(\text{(function template specialization signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced)}\)

3.25 [defns.signature.member]
signature
\(\text{(class member function) name, parameter-type-list (9.3.3.5), class of which the function is a member, }\)
\(\text{cv-qualifiers (if any), ref-qualifier (if any), and trailing requires-clause (9.3) (if any)}\)

3.26 [defns.signature.member.templ]
signature
\(\text{(class member function template) name, parameter-type-list (9.3.3.5), class of which the function is a member, }\)
\(\text{cv-qualifiers (if any), ref-qualifier (if any), return type (if any), template-head, and trailing requires-clause (9.3) (if any)}\)

3.27 [defns.signature.member.spec]
signature
\(\text{(class member function template specialization signature of the member function template of which it is a specialization and its template arguments (whether explicitly specified or deduced)}\)

3.28 [defns.static.type]
static type
type of an expression (6.8) resulting from analysis of the program without considering execution semantics

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

3.29 [defns.unblock]
unblock
satisfy a condition that one or more blocked threads of execution are waiting for

3.30 [defns.undefined]
undefined behavior
behavior for which this document imposes no requirements

\[\text{[Note 1 to entry: Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression never exhibits behavior explicitly specified as undefined in Clause 4 through Clause 15 of this document (7.7). — end note]}\]

3.31 [defns.unspecified]
unspecified behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation

\[\text{[Note 1 to entry: The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this document. — end note]}\]
3.32 well-formed program

C++ program constructed according to the syntax rules, diagnosable semantic rules, and the one-definition rule (6.3)
4 General principles

4.1 Implementation compliance

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 this document, a conforming implementation shall, within its resource limits, accept and correctly execute the 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: During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 13.10.2. — 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 (16.5.2.2).

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 (16.5.1.3).

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.3

4.1.1 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 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.4

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

3) This documentation also defines implementation-defined behavior; see 4.1.1.

4) 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.

§ 4.1.1
Certain aspects and operations of the abstract machine are described in this document as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects. 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 (7.6.1.2)). 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. Certain other operations are described in this document as undefined (for example, the effect of attempting to modify a const object). [Note: This document imposes no requirements on the behavior of programs that contain undefined behavior. — end note]

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).

The least requirements on a conforming implementation are:

1. Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.
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.
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.

These collectively are referred to as the observable behavior of the program. [Note: More stringent correspondences between abstract and actual semantics may be defined by each implementation. — end note]

### 4.2 Structure of this document

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. 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.

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

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.

Throughout this document, each example is introduced by “[Example: ” and terminated by “— end example]”. Each note is introduced by “[Note: ” or “[Note n to entry: ” and terminated by “— end note]”. Examples and notes may be nested.

### 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

\[
\{ \text{expression}_{\text{opt}} \}
\]

5) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 4.1.
indicates an optional expression enclosed in braces.

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

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

4.4 Acknowledgments


Portions of the library Clauses of this document are based on work by P.J. Plauger, which was published as *The Draft Standard C++ Library* (Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger).

POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

ECMAScript® is a registered trademark of Ecma International.

Unicode® is a registered trademark of Unicode, Inc.

All rights in these originals are reserved.
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 (16.5.1.2) 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: A C++ program need not all be translated at the same time. —end note]

2 [Note: 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.6

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 white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.7 Each comment is replaced by one space character. New-line characters are retained. Whether each non-empty sequence of white-space 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: 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 (15.6.3), 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.8

---

6) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

7) 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 * or >. A partial comment would arise from a source file ending with an unclosed /* comment.

8) 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: The process of analyzing and translating the tokens may 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: 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: Some or all of these may be supplied from a library. —end note] Each translated translation unit is examined to produce a list of required instantiations. [Note: This may include instantiations which have been explicitly requested (13.9.2). —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: An implementation could 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: 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 [lex.charset]

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:9

\[
\begin{align*}
\text{abcdefgijklmnopqrstuvwxyz} \\
\text{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\
0123456789 \\
\text{ _ } \\
\end{align*}
\]

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

\[
\text{hex-quad:} \\
\begin{align*}
\text{universal-character-name:} \\
& \text{hex-quad} \\
& \text{hex-quad hex-quad hex-quad hex-quad}
\end{align*}
\]

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.10 [Note: ISO/IEC 10646 code points are integers in the range [0, 10FFFF] (hexadecimal). A surrogate code point is a value in the range [D800, DFFF] (hexadecimal). A control character is a character whose code point is in either of the ranges [0, 1F] or [7F, 9F] (hexadecimal). —end note]

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

---

9) 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, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.

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

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-white-space 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 white space; this consists of comments (5.7), or white-space 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, white space (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:

#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: None has any observable spelling. —end note]

[Example: 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: 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

Alternative token representations are provided for some operators and punctuators.¹¹

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.¹² The set of alternative tokens is defined in Table 1.

Table 1: Alternative tokens 

<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><code>&lt;%</code></td>
<td>{</td>
<td><code>and</code></td>
<td><code>&amp;&amp;</code></td>
<td><code>and_eq</code></td>
<td><code>&amp;=</code></td>
</tr>
<tr>
<td><code>%&gt;</code></td>
<td>}</td>
<td><code>bitor</code></td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>&lt;:</code></td>
<td>[</td>
<td><code>or</code></td>
<td>`</td>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>:</code></td>
<td>]</td>
<td><code>xor</code></td>
<td><code>~</code></td>
<td><code>not_eq</code></td>
<td><code>!=</code></td>
</tr>
<tr>
<td><code>%</code>:</td>
<td>#</td>
<td><code>compl</code></td>
<td><code>~</code></td>
<td><code>not_eq</code></td>
<td><code>!=</code></td>
</tr>
<tr>
<td><code>%:%:</code></td>
<td><code>##</code></td>
<td><code>bitand</code></td>
<td><code>&amp;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5.6 Tokens

There are five kinds of tokens: identifiers, keywords, literals,¹³ operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, “white space”), as described below, are ignored except as they serve to separate tokens. [Note: Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. —end note]

### 5.7 Comments

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 white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [Note: 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]

---

¹¹ 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”.

¹² Thus the “stringized” values (15.6.2) of `|` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

¹³ Literals include strings and character and numeric literals.
5.8 Header names

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 "

[Note: 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.

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

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

14) 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.

Table 2: Ranges of characters allowed

<table>
<thead>
<tr>
<th>00A8</th>
<th>00AA</th>
<th>00AD</th>
<th>00AF</th>
<th>00B2-00B5</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B7-00BA</td>
<td>00BC-00BE</td>
<td>00C0-00D6</td>
<td>00D8-00F6</td>
<td>00F8-00FF</td>
</tr>
<tr>
<td>0100–167F</td>
<td>1681–180D</td>
<td>180F–1FFF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2070–218F</td>
<td>2460–24FF</td>
<td>2776–2793</td>
<td>2C00–2DFF</td>
<td>2E80–2FFF</td>
</tr>
<tr>
<td>3004–3007</td>
<td>3021–302F</td>
<td>3031–D7FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F900–FD3D</td>
<td>FD40–FDFF</td>
<td>FDF0–FE44</td>
<td>FE47–FFFD</td>
<td></td>
</tr>
<tr>
<td>10000–1FFFD</td>
<td>20000–2FFFD</td>
<td>30000–3FFFD</td>
<td>40000–4FFFD</td>
<td>50000–5FFFD</td>
</tr>
<tr>
<td>60000–6FFFD</td>
<td>70000–7FFFD</td>
<td>80000–8FFFD</td>
<td>90000–9FFFD</td>
<td>A0000–AFFFD</td>
</tr>
<tr>
<td>B0000–BFFFD</td>
<td>C0000–CFFFD</td>
<td>D0000–DFFFD</td>
<td>E0000–EFFFD</td>
<td></td>
</tr>
</tbody>
</table>

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

| 0300–036F | 1DC0–1DFF | 20D0–20FF | PE20–FE2F |

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

| final | import | module | override |

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 |

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

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

15) On systems in which linkers cannot accept extended characters, an encoding of the universal-character-name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the \u in a universal-character-name. Extended characters may 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.

§ 5.11
5.12 Operators and punctuators

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-punct>
  <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 or or_eq xor xor_eq
```

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

5.13 Literals

5.13.1 Kinds of literals

There are several kinds of literals.\(^{16}\)

\(^{16}\) The term “literal” generally designates, in this document, those tokens that are called “constants” in ISO C.
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

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 7; the lexically first digit of the sequence of digits is the most significant. [Note: The prefix and any optional separating single quotes are ignored when determining the value. — end note]
2 The hexadecimal-digits a through f and A through F have decimal values ten through fifteen. [Example: The number twelve can be written 12, 014, 0XC, or 0b1100. The integer-literals 1048576, 1’048’576, 0x1000000, 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.

4 If an integer-literal cannot be represented by any type in its list and an extended integer type (6.8.1) 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 [lex.ccon]

character-literal:
  encoding-prefixopt ' c-char-sequence '

echoading-prefix: one of
  u8 u U L

c-char-sequence: one of
  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
  \'  \"  \?  \\  \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

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: 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: 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.17 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: The type wchar_t is able to represent all members of the execution wide-character set (see 6.8.1). —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 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: If the value

17) They are intended for character sets where a character does not fit into a single byte.
18) Using an escape sequence for a question mark is supported for compatibility with ISO C++ 2014 and ISO C.
Table 9: Escape sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line NL(LF)</td>
<td>\n</td>
</tr>
<tr>
<td>horizontal tab</td>
<td>HT \t</td>
</tr>
<tr>
<td>vertical tab</td>
<td>VT \v</td>
</tr>
<tr>
<td>backspace</td>
<td>BS \b</td>
</tr>
<tr>
<td>carriage return</td>
<td>CR \r</td>
</tr>
<tr>
<td>form feed</td>
<td>FF \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</td>
<td>&quot; \</td>
</tr>
<tr>
<td>octal number</td>
<td>ooo \ooo</td>
</tr>
<tr>
<td>hex number</td>
<td>hhh \xhh</td>
</tr>
</tbody>
</table>

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: 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 may use its own native character set, so long as the same results are obtained. —end note]

5.13.4 Floating-point literals

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
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
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 l F L

1 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 [tab:lex.fcon.type]

<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. \[Note: Any optional separating single quotes are ignored when determining the value. — end note\] 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. \[Example: The floating-point-literals 49.625 and 0xC.68p+2 have the same value. The floating-point-literals 1.602176565e-19 and 1.602176565e-19 have the same value. — end example\]

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 [lex.string]

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.

\[Note: The characters '(' and ')' are permitted in a raw-string. Thus, R"delimiter((a|b))delimiter" is equivalent to "(a|b)". — end note\]

\[Note: 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:
const char* p = R"(a\nb\nc)";
assert(std::strcmp(p, "a\nb\nc") == 0);

— end note

4 [Example: The raw string

R"a(\n)\na"

is equivalent to "a\na". The raw string

R"(x = "y")"

is equivalent to "x = "y"". — end example]

5 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.

6 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.

8 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: 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.

10 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.

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.

11 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: 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 11: String literal concatenations [tab:lex.string.concat]

<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>u&quot;b&quot;</td>
</tr>
</tbody>
</table>

Characters in concatenated strings are kept distinct.

[Example: 

"\x4A" "B"

§ 5.13.5
contains the two characters \xA and \B after concatenation (and not the single hexadecimal character \xAB).

— end example

After any necessary concatenation, in translation phase 7 (5.2), ‘\0’ is appended to every string-literal so that programs that scan a string can find its end.

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 char32_t or wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating U'\0' or L'\0'. 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'\0'. [Note: The size of a char16_t string literal is the number of code units, not the number of characters. — end note] [Note: 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 \0.

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

5.13.6  Boolean literals

boolean-literal:
false
true

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

The pointer literal is the keyword nullptr. It is a prvalue of type std::nullptr_t. [Note: 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 7.3.11 and 7.3.12. — 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
user-defined-string-literal:
string-literal ud-suffix
user-defined-character-literal:
character-literal ud-suffix

§ 5.13.8 23
If a token matches both user-defined-literal and another literal kind, it is treated as the latter. [Example: 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.

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 (6.5.1). Let S be the set of declarations found by this lookup. S shall not be empty.

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

\[
\text{operator} \ "X\text{(nULL)}
\]

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

\[
\text{operator} \ "X\text{"n"}
\]

Otherwise (S contains a numeric literal operator template), L is treated as a call of the form

\[
\text{operator} \ "X\text{<'}c_1\', 'c_2', ... 'c_k'>()}
\]

where n is the source character sequence c1c2...ck. [Note: The sequence c1c2...ck can only contain characters from the basic source character set. —end note]

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

\[
\text{operator} \ "X\text{(fL)}
\]

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

\[
\text{operator} \ "X\text{"f"}
\]

Otherwise (S contains a numeric literal operator template), L is treated as a call of the form

\[
\text{operator} \ "X\text{<'}c_1\', 'c_2', ... 'c_k'>()}
\]

where f is the source character sequence c1c2...ck. [Note: The sequence c1c2...ck can only contain characters from the basic source character set. —end note]

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

\[
\text{operator} \ "X\text{<str>}
\]

Otherwise, the literal L is treated as a call of the form

\[
\text{operator} \ "X\text{(str, len)}
\]

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

\[
\text{operator} \ "X\text{(ch)}
\]

[Example:

\[
\begin{align*}
\text{long double operator "}_w\text{(long double);}
\text{std::string operator "}_w\text{(const char16_t*, std::size_t);}
\text{unsigned operator "}_w\text{(const char*);}
\text{int main() \{}
\text{1.2}_w; \quad \text{// calls operator "}_w\text{(1.2L)}
\text{u"one"}_w; \quad \text{// calls operator "}_w\text{(u"one", 3)}
\text{12}_w; \quad \text{// calls operator "}_w\text{("12")}
\text{"two"}_w; \quad \text{// error: no applicable literal operator}
\text{\}}
\end{align*}
\]

—end example]
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:

```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: 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: 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 (11.4.7.2), or template-id (13.3) that denotes an entity or label (8.7.5, 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 (8.7.5) 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 (6.7.5.3), 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 (13.9.2),
— guiding template argument deduction for constructors (13.7.1.2),
— 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),
(2.2) — it contains the 
\textbf{extern} specifier (9.2.1) or a \textit{linkage-specification}\(^{19}\) (9.11) and neither an \textit{initializer} nor a \textit{function-body},

(2.3) — it declares a non-inline static data member in a class definition (11.4, 11.4.8),

(2.4) — it declares a static data member outside a class definition and the variable was defined within the class with the \texttt{constexpr} specifier (this usage is deprecated; see D.6),

(2.5) — it is introduced by an \textit{elaborated-type-specifier} (11.3),

(2.6) — it is an \textit{opaque-enum-declaration} (9.7.1),

(2.7) — it is a \textit{template-parameter} (13.2),

(2.8) — it is a \textit{parameter-declaration} (9.3.3.5) in a function declarator that is not the \textit{declarator} of a \textit{function-definition},

(2.9) — it is a \texttt{typedef} declaration (9.2.3),

(2.10) — it is an \textit{alias-declaration} (9.2.3),

(2.11) — it is a \textit{using-declaration} (9.9),

(2.12) — it is a \textit{deduction-guide} (13.7.1.2),

(2.13) — it is a \textit{static_assert-declaration} (9.1),

(2.14) — it is an \textit{attribute-declaration} (9.1),

(2.15) — it is an \textit{empty-declaration} (9.1),

(2.16) — it is a \textit{using-directive} (9.8.3),

(2.17) — it is a \textit{using-enum-declaration} (9.7.2),

(2.18) — it is a \textit{template-declaration} (13.1) whose \textit{template-head} is not followed by either a \textit{concept-definition} or a \textit{declaration} that defines a function, a class, a variable, or a static data member.

(2.19) — it is an explicit instantiation declaration (13.9.2), or

(2.20) — it is an explicit specialization (13.9.3) whose \textit{declaration} is not a definition.

A declaration is said to be a \textit{definition} of each entity that it defines. \textit{Example:} All but one of the following are definitions:

\begin{verbatim}
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(int x): x(x) {}  // 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
\end{verbatim}

whereas these are just declarations:

\begin{verbatim}
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
typedef X anotherX;  // declares anotherX
using N::d;  // declares d
\end{verbatim}

\textit{— end example}\(^{19}\)

\(^{19}\) Appearing inside the brace-enclosed \texttt{declaration-seq} in a \textit{linkage-specification} does not affect whether a declaration is a definition.
[Note: In some circumstances, C++ implementations implicitly define the default constructor (11.4.4.1), copy constructor, move constructor (11.4.4.2), copy assignment operator, move assignment operator (11.4.5), or destructor (11.4.6) member functions. — end note] [Example: Given]

```cpp
#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

```cpp
struct C {
    std::string s;
    C() : s() { }
    C(const C& x): s(x.s) { }  // : s(std::move(x.s)) { }
    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() { }
};
```

—end example]
A function is named by an expression or conversion as follows:

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 (7.6.2.1). [Note: This covers taking the address of functions (7.3.3, 7.6.2.1), calls to named functions (7.6.1.2), operator overloading (Clause 12), user-defined conversions (11.4.7.2), allocation functions for new-expression, 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 (11.10.5). — end note]

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 7.6.2.7.

3. A deallocation function for a class is named by a delete-expression if it is the selected usual deallocation function as specified in 7.6.2.8 and 11.12.

4. 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 (7.3.1) 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.

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

6. *this is odr-used if this appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (11.4.2)).

7. 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 (11.4.5).

8. 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 11.4.5. 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 (11.4.6).

9. A local entity (6.1) is odr-usable in a declarative region (6.4.1) if:
   (9.1) either the local entity is not *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 *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

20) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.
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:

```c
void f(int n) {
  [] { n = 1; }; // error: n is not odr-usable due to intervening lambda-expression
  struct A {
    void f() { n = 2; } // error: n is not odr-usable due to intervening function definition scope
  };
  void g(int = n);
  [=](int k = n) {}; // error: n is not odr-usable due to being outside the block scope of the lambda-expression
  [k] { [n]{ return n; }; }; // OK
}
```
—end example]

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 (8.5.1); 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 11.4.4.1, 11.4.4.2, 11.4.6, and 11.4.5). [Example:

```c
auto f() {
  struct A {};
  return A();
}
dcltype(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]

11 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.

12 A definition of a class is required to be reachable in every context in which the class is used in a way that requires the class type to be complete. [Example: The following complete translation unit is well-formed, even though it never defines X:

```c
struct X; // declare X as a struct type
struct X* x1; // use X in pointer formation
X* x2; // use X in pointer formation
```
—end example] [Note: 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 (7.6.2.7), or
(12.4) an ivalue-to-rvalue conversion is applied to a glvalue referring to an object of type T (7.3.1), or
(12.5) an expression is converted (either implicitly or explicitly) to type T (7.3, 7.6.1.3, 7.6.1.6, 7.6.1.8, 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 (7.6.1.6) or a static_cast (7.6.1.8), or
(12.7) a class member access operator is applied to an expression of type T (7.6.1.4), or
(12.8) the typeid operator (7.6.1.7) or the sizeof operator (7.6.2.4) 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 (7.6.1.2), or
(12.10) a class with a base class of type T is defined (11.7), or

§ 6.3 30
an lvalue of type \( T \) is assigned to (7.6.19), or

- the type \( T \) is the subject of an \texttt{alignof} expression (7.6.2.5), or

- an \texttt{exception-declaration} has type \( T \), reference to \( T \), or pointer to \( T \) (14.4).

\end{quote}

There can be more than one definition of a

- class type (Clause 11),
- enumeration type (9.7.1),
- inline function or variable (9.2.7),
- templated entity (13.1),
- default argument for a parameter (for a function in a given scope) (9.3.3.6), 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 \texttt{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 (13.10.3), 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 \texttt{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) \texttt{template-id} or \texttt{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 (11.4.4.1, 11.4.4.2), 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:

```c
// translation unit 1:
struct X {
    X(int, int);
    X(int, int, int);
};
X::X(int, int = 0) {}
class D {
    X x = 0;
};
```
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 \( D \).

14 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 (13.8.3), and also to dependent names at the point of instantiation (13.8.2). 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: 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 \( D \) may result in the different declarations having distinct types, and lambda-expressions appearing in a default argument of \( D \) may 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:

```cpp
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]

17 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: In

```c
int j = 24;
int main() {
    int i = j, j;
    j = 42;
}
```

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

—end example

The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a friend specifier (11.9.3), certain uses of the elaborated-type-specifier (9.2.8.3), and using-directives (9.8.3) alter this general behavior.

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

1. they shall all refer to the same entity, or all refer to functions and function templates; or
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: 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: 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, elaborated-type-specifiers (9.2.8.3) and friend declarations (11.9.3) may introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (6.6) may 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]

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

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

### 6.4.2 Point of declaration

The point of declaration for a name is immediately after its complete declarator (9.3) and before its initializer (if any), except as noted below. [Example:

```c
unsigned char x = 12;
{ unsigned char x = x; }
```

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

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

```c
const int i = 2;
{ int i[1]; }
```

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

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

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

5 The point of declaration for an enumerator is immediately after its enumerator-definition. [Example:
\[
\begin{aligned}
    \text{const int } x &= 12; \\
    \{ \text{enum } \{ x = x \}; \}
\end{aligned}
\]
Here, the enumerator \( x \) is initialized with the value of the constant \( x \), namely 12. — end example]

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

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

(7.1) — for a declaration of the form
\[
\begin{aligned}
    \text{class-key attribute-specifier-seq}_{\text{opt}} \text{ identifier } ;
\end{aligned}
\]
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
\[
\begin{aligned}
    \text{class-key identifier} \\
    \text{if the elaborated-type-specifier is used in the decl-specifier-seq or parameter-declaration-clause of a function defined in namespace scope, the identifier is declared as a class-name in the namespace that contains the declaration; otherwise, except as a friend declaration, the identifier is declared in the smallest namespace or block scope that contains the declaration. [Note: These rules also apply within templates. — end note] [Note: Other forms of elaborated-type-specifier do not declare a new name, and therefore must refer to an existing type-name. See 6.5.4 and 9.2.8.3. — end note]}
\]

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 (8.6.4) is immediately after the for-range-initializer.

12 The point of declaration for a template parameter is immediately after its complete template-parameter.
[Example:
\[
\begin{aligned}
    \text{typedef unsigned char } T; \\
    \text{template<class } T \\
    = T \quad // \text{lookup finds the typedef name of unsigned char} \\
    , T \quad // \text{lookup finds the template parameter} \\
    N = 0\text{> struct } A \{ \};
\end{aligned}
\]
—end example]

13 [Note: 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 (9.8.1.2). 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]

14 [Note: For point of instantiation of a template, see 13.8.4.1. — 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:

```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 (9.8.3) 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:

```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 1+a;   // l is from unnamed namespace
   }
   int i;         // error: duplicate definition
   int j();       // OK: duplicate function declaration
```]

§ 6.4.6
© ISO/IEC

N4861

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 module-import-declaration or 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 module-declaration in Q (if any), and
(2.3) — either X is exported or Q and R are part of the same module.

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

Translation unit #1:
export module Q;
export int sq(int i) { return i*i; }

Translation unit #2:
export module R;
export import Q;

Translation unit #3:
import R;
int main() { return sq(9); } // OK: sq from module Q
— end example] — end note]

3 A namespace member can also be referred to after the :: scope resolution operator (7.5.4.2) applied to the name of its namespace or the name of a namespace which nominates the member’s namespace in a using-directive; see 6.5.3.2.

4 The outermost declarative region of a translation unit is also a namespace, called the global namespace. A name declared in the global namespace has global namespace scope (also called 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 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 declarator-id, including a parameter-declaration-clause and any default arguments (9.3.3.6)).

[Example:

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

§ 6.4.7 36
The name of a class member shall only be used as follows:

- (6.1) in the scope of its class (as described above) or a class derived (11.7) from its class,
- (6.2) after the . operator applied to an expression of the type of its class (7.6.1.4) or a class derived from its class,
- (6.3) after the -> operator applied to a pointer to an object of its class (7.6.1.4) or a class derived from its class,
- (6.4) after the :: scope resolution operator (7.5.4.2) applied to the name of its class or a class derived from its class.

### 6.4.8 Enumeration scope

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

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.

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:**

```cpp
namespace N {
  template<class T> struct A { }; // #1
  template<class U> void f(U) { } // #2
  struct B {
    template<class V> friend int g(struct C*); // #3
  }
}
```

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*

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: 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,

```cpp
template<class T, T* p, class U = T> class X { /* ... */ }; // #1
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,

```cpp
template<class T> class X : public Array<T> { /* ... */ }; // #2
```
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: As a result, a template-parameter hides any entity with the same name in an enclosing scope (6.4.10). [Example:

```
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: Because the name of a template parameter cannot be redeclared within its potential scope (13.8.1), a template parameter’s scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 13.8.1. — 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 6.5.1.

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 (6.5.1)), 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 6.5.3.2.

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

6.5 Name lookup [basic.lookup]

1 The name lookup rules apply uniformly to all names (including typedef-names (9.2.3), 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: 6.6 discusses linkage issues. The notions of scope, point of declaration and name hiding are discussed in 6.4. — end note]

6.5.1 Unqualified name lookup [basic.lookup.unqual]

1 In all the cases listed in 6.5.1, 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.

2 The declarations from the namespace nominated by a using-directive become visible in a namespace enclosing the using-directive; see 9.8.3. For the purpose of the unqualified name lookup rules described in 6.5.1, the declarations from the namespace nominated by the using-directive are considered members of that enclosing namespace.
The lookup for an unqualified name used as the *postfix-expression* of a function call is described in 6.5.2.  [Note: For purposes of determining (during parsing) whether an expression is a *postfix-expression* for a function call, the usual name lookup rules apply. In some cases a name followed by `<` is treated as a *template-name* even though name lookup did not find a *template-name* (see 13.3). For example,

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

The rules in 6.5.2 have no effect on the syntactic interpretation of an expression. For example,

```c
typedef int f;
namespace N {
  struct A {
    friend void f(A &);
    operator int();
    void g(A a) {
      int i = f(a);
      // 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 (6.5.2) does not apply and the friend function *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 *N*, a name used after the function’s *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 *N* or, if *N* is a nested namespace, shall be declared before its use in one of *N*’s enclosing namespaces.  [Example:

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

—*end example*]

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:

(7.1) — before its use in class *X* or be a member of a base class of *X* (11.8), or

---

21) This refers to unqualified names that occur, for instance, in a type or default argument in the *parameter-declaration-clause* or used in the function body.

22) This refers to unqualified names following the class name; such a name may be used in a *base-specifier* or in the *member-specification* of the class definition.
— if \( X \) is a nested class of class \( Y \) (11.4.10), 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),\(^{23}\) or

— 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

— 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:

```cpp
namespace M {
    class B { };
}

namespace N {
    class Y : public M::B {
        class X {
            int a[i];
        };
    };
}

// 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
```

— end example]  

[Note: 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 9.8.1.2. — end note]  

[Note: 6.4.7 further describes the restrictions on the use of names in a class definition. 11.4.10 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]

8 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\(^{24}\), shall be declared in one of the following ways:

— before its use in the block in which it is used or in an enclosing block (8.4), or

— shall be a member of class \( X \) or be a member of a base class of \( X \) (11.8), or

— if \( X \) is a nested class of class \( Y \) (11.4.10), 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),\(^{25}\) or

— 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

— 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.

[Example:

```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: 11.4.1 and 11.4.8 further describe the restrictions on the use of names in member function definitions. 11.4.10 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 (11.9.3) 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.

In a friend declaration naming a member function, a name used in the function declarator and not part of a template-argument in the declarator-id in the declarator-id is first looked up in the scope of the member function’s class (11.8). If it is not found, or if the name is part of a template-argument in the declarator-id, the look up is as described for unqualified names in the definition of the class granting friendship. [Example:

```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]

11 During the lookup for a name used as a default argument (9.3.3.6) in a function parameter-declaration-clause or used in the expression of a mem-initializer for a constructor (11.10.2), 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: 9.3.3.6 further describes the restrictions on the use of names in default arguments. 11.10.2 further describes the restrictions on the use of names in a ctor-initializer. — end note]

12 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.

13 A name used in the definition of a static data member of class X (11.4.8.2) (after the qualified-id of the static member) is looked up as if the name was used in a member function of X. [Note: 11.4.8.2 further describes the restrictions on the use of names in the definition of a static data member. — end note]

14 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:

```cpp
namespace N {
    int i = 4;
    extern int j;
}
```
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: But function parameter names are found. — end note]

15

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

6.5.2 Argument-dependent name lookup [basic.lookup.argdep]

1 When the postfix-expression in a function call (7.6.1.2) is an unqualified-id, other namespaces not considered during the usual unqualified lookup (6.5.1) may be searched, and in those namespaces, namespace-scope friend function or function template declarations (11.9.3) 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:

```cpp
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
}
```

— end example]

2 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:

1. If \( T \) is a fundamental type, its associated sets of namespaces and entities are both empty.
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: Non-type template arguments do not contribute to the set of associated namespaces. — end note]
3. 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.
4. If \( T \) is a pointer to \( U \) or an array of \( U \), its associated namespaces and entities are those associated with \( U \).
5. 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.
6. 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 \( X \).
7. 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 (9.8.1), 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 (6.5.1) and let \( Y \) be the lookup set produced by argument dependent lookup (defined as follows). If \( X \) contains

1. a declaration of a class member, or
2. a block-scope function declaration that is not a using-declaration, or
3. 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: The namespaces and entities associated with the argument types can include namespaces and entities already considered by the ordinary unqualified lookup. — end note]  

Example:

```cpp
namespace NS {
    class T { }
    void f(T);
    void g(T, int);
}
NS::T parm;
void g(NS::T, float);
int main() {
    f(parm);  // OK: calls NS::f
    extern void g(NS::T, float);
    g(parm, 1);  // OK: calls g(NS::T, float)
}
```

Example:

5. When considering an associated namespace \( N \), the lookup is the same as the lookup performed when \( N \) is used as a qualifier (6.5.3.2) except that:

1. Any using-directives in \( N \) are ignored.
2. All names except those of (possibly overloaded) functions and function templates are ignored.
3. Any namespace-scope friend functions or friend function templates (11.9.3) 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 (9.8.1.2).
4. 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 \( D \).
5. If the lookup is for a dependent name (13.8.2, 13.8.4.2), any declaration \( D \) in \( N \) is visible if \( D \) would be visible to qualified name lookup (6.5.3.2) 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:

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]

### 6.5.3 Qualified name lookup

The name of a class or namespace member or enumerator can be referred to after the `::` scope resolution operator (§7.5.4.2) 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:

```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]

2 [Note: Multiply qualified names, such as `N1::N2::N3::n`, can be used to refer to members of nested classes (11.4.10) or members of nested namespaces. —end note]

3 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:

```cpp
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 ::X C::arr[C::number];
```

—end example]
A name prefixed by the unary scope operator :: (7.5.4.2) 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 (6.5.3.2). 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:

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

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

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

- end example] [Note: 6.5.5 describes how name lookup proceeds after the , and -> operators. — end note]

### 6.5.3.1 Class members [class.qual]

1 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: 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:

(1.1) — the lookup for a destructor is as specified in 6.5.3;

(1.2) — 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 6.5.5);

(1.3) — the names in a template-argument of a template-id are looked up in the context in which the entire postfix-expression occurs;

(1.4) — 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).

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

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

(2.2) — 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: 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:

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

26) Lookups in which function names are ignored include names appearing in a nested-name-specifier, an elaborated-type-specifier, or a base-specifier.
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]
// 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,
// 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: The same declaration found more than once is not an ambiguity (because it is still a unique declaration).

Example:

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]

5 [Example: 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]

6 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:

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]

7 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

    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 (9.8.1) of that namespace. [Example:

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:

namespace A {
    namespace B {
        void f1(int);
    }
}
6.5.4 Elaborated type specifiers

An elaborated-type-specifier (9.2.8.3) 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:

```
class-key attribute-specifier-seq_opt identifier ;
```

the identifier is looked up according to 6.5.1 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:

```
class-key attribute-specifier-seq_opt 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 6.5.3, 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:

```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 (9.2.8.3)
    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 (9.2.8.3)
struct Base::Data;        // error: cannot introduce a qualified type (9.2.8.3)
struct Base::Data* thisBase; // error: Datum undefined
struct Base::Data* pBase; // OK: refers to nested Data
```

— end example]
6.5.5 Class member access

In a class member access expression (7.6.1.4), if the . or -> token is immediately followed by an identifier followed by a <, the identifier must be 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.

If the id-expression in a class member access (7.6.1.4) 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).

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:

```cpp
struct A { 
};

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

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

—end example]

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: See 6.5.3, which describes the lookup of a name before ::, which will only find a type or namespace name. — end note]

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.

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.

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:

```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]

6.5.6 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-seq_opt
  global-module-fragment_opt
  module-declaration declaration-seq_opt
  private-module-fragment_opt
```

A name is said to have linkage when it might 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.

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
3. A data member of an anonymous union.

[Note: An instantiated variable template that has const-qualified type can have external or module linkage, even if not declared extern. — end note]

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:

1. A variable; or
2. A function; or
3. 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 (9.2.3); or
4. 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 (9.2.3); or
5. An unnamed enumeration that has an enumerator as a name for linkage purposes (9.7.1); or
6. A template

has its linkage determined as follows:

1. If the enclosing namespace has internal linkage, the name has internal linkage;
2. 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;
3. 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 (9.2.3), 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:

```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]

7 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:

```c
namespace X {
    void p() {
        q();           // error: q not yet declared
        extern void q();  // q is a member of namespace X
    }

    void middle() {
        q();           // error: q not yet declared
    }

    void q() { /* ... */ }       // definition of X::q
}
```

—end example]

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

- 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
- both names refer to members of the same namespace or to members, not by inheritance, of the same class; and
- when both names denote functions or function templates, the signatures (3.20, 3.22) 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: 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 may serve as the name of an enumeration with linkage (9.7.1). —end note]

10 If a declaration would redefine a reachable declaration attached to a different module, the program is ill-formed. [Example:

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

After all adjustments of types (during which typedefs (9.2.3) 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 (9.3.3.4). A violation of this rule on type identity does not require a diagnostic.

[Note: Linkage to non-C++ declarations can be achieved using a linkage-specification (9.11). — end note]

A declaration $D$ names an entity $E$ if

1. $D$ contains a lambda-expression whose closure type is $E$,
2. $E$ 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
3. $E$ 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: Non-dependent names in an instantiated declaration do not refer to a set of overloads (13.8.3). — end note]

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

1. 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 (9.2.8.5)),
2. the initializer for a variable or variable template (but not the variable’s type),
3. friend declarations in a class definition, and
4. 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: 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

1. a type, function, variable, or template that
   1. has a name with internal linkage, or
   2. does not have a name with linkage and is declared, or introduced by a lambda-expression, within the definition of a TU-local entity,
2. 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,
3. a specialization of a TU-local template,
4. a specialization of a template with any TU-local template argument, or
5. a specialization of a template whose (possibly instantiated) declaration is an exposure. [Note: The specialization might have been implicitly or explicitly instantiated. — end note]

A value or object is TU-local if either

1. it is, or is a pointer to, a TU-local function or the object associated with a TU-local variable,
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.7).

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 (13.8.4.1).

[Example:

Translation unit #1:

```cpp
export module A;
static void f() {} // error: is an exposure of f
inline void it() { f(); } // OK
template<int> void g() { its(); } // OK
template void g<>() {};

decltype(f) *fp; // error: f (though not its type) is TU-local
auto &fr = f; // OK
constexpr auto &fr2 = fr; // 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:

```cpp
module A;
void other() {
  g<>(); // OK, specialization is explicitly instantiated
g<>(); // error: instantiation 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 [basic.memobj]

6.7.1 Memory model [intro.memory]

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

27) The number of bits in a byte is reported by the macro CHAR_BIT in the header <climits> (17.3.6).
bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

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

3 A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width. [Note: 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] Two or more threads of execution (6.9.2) can access separate memory locations without interfering with each other.

4 [Note: 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]

5 [Example: A class declared as

```cpp
struct {
    char a;
    int b:5,
    c:11,
    :0,
    d:8;
    struct {int ee:8;} e;
}
```

contains four separate memory locations: The member a and bit-fields d and e.ee are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields b and c together constitute the fourth memory location. The bit-fields b and c cannot be concurrently modified, but b and a, for example, can be. — end example]

6.7.2 Object model [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is created by a definition (6.2), by a new-expression (7.6.2.7), 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 (7.3.4, 6.7.7). An object occupies a region of storage in its period of construction (11.10.4), throughout its lifetime (6.7.3), and in its period of destruction (11.10.4). [Note: 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 (11.7.2); 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 expressions (7.6) used to access them.

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

(2.1) — the lifetime of e’s containing object has begun and not ended, and

(2.2) — the storage for the new object exactly overlays the storage location associated with e, and

(2.3) — the new object is of the same type as e (ignoring cv-qualification).

3 If a complete object is created (7.6.2.7) in storage associated with another object e of type “array of N unsigned char” or of type “array of N std::byte” (17.2.1), that array provides storage for the created object if:

(3.1) — the lifetime of e has begun and not ended, and

(3.2) — the storage for the new object fits entirely within e, and

(3.3) — there is no smaller array object that satisfies these constraints.

§ 6.7.2
Note: 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:

```cpp
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();
    return *c + *d; // OK
}
```

Example:

```cpp
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:

- \( a \) is a subobject of \( b \), or
- \( b \) provides storage for \( a \), or
- 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:

- If \( x \) is a complete object, then the complete object of \( x \) is itself.
- Otherwise, the complete object of \( x \) is the complete object of the (unique) object that contains \( x \).

6 If a complete object, a data member (11.4), 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:
- a base class subobject, or
- a non-static data member declared with the `no_unique_address` attribute (9.12.10).

8 An object has nonzero size if it

- is not a potentially-overlapping subobject, or
- is not of class type, or
- is of a class type with virtual member functions or virtual base classes, or
- has subobjects of nonzero size or 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 (11.4.9), 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.28 [Example:

```cpp
static const char test1 = 'x';
```

28) 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).
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.

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. [Note: Such operations do not start
the lifetimes of subobjects of such objects that are not themselves of implicit-lifetime types. — end note]

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:
#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 (6.8.2) 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;
}
— end example]

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: 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: Some functions in the C++ standard library implicitly create objects (20.10.9.2, 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:

(1.1) — storage with the proper alignment and size for type T is obtained, and
(1.2) — 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 (9.4.1, 11.10.2), or as described in 11.5 and 11.4.4.2, and except as
described in 20.10.10.1. The lifetime of an object o of type T ends when:

(1.3) — if T is a non-class type, the object is destroyed, or
(1.4) — if T is a class type, the destructor call starts, or
(1.5) — 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.
The properties ascribed to objects and references throughout this document apply for a given object or reference only during its lifetime. [Note: 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 11.10.2 and in 11.10.4. 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 ended. 11.10.2 and 11.10.4 describe the behavior of an object during its periods of construction and destruction. — end note]

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 (7.5.4.3) 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 (7.6.2.8) 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 11.10.4. Otherwise, such a pointer refers to allocated storage (6.7.5.4.1), 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 lvalue 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, and
3. The pointer is implicitly converted (7.3.11) to a pointer to a virtual base class, or
4. The pointer is used as the operand of a static_cast (7.6.1.8), except when the conversion is to pointer 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 (7.6.1.6).

Example:
```c++
#include <cstdlib>

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

struct D1 : B { void f(); };
struct D2 : B { void f(); };

void B::mutate() {
    new (this) D2; // reuses storage — ends the lifetime of *this
    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
}
```

29) For example, before the construction of a global object that is initialized via a user-provided constructor (11.10.4).
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 11.10.4. Otherwise, such a glvalue refers to allocated storage (6.7.5.4.1), and using the properties of the glvalue that do not depend on its value is well-defined. The program has undefined behavior if:

7.1 — the glvalue is used to access the object, or
7.3 — the glvalue is bound to a reference to a virtual base class (9.4.3), or
7.4 — the glvalue is used as the operand of a dynamic_cast (7.6.1.6) or as the operand of typeid.

8 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:

8.1 — the storage that \( o_2 \) occupies exactly overlays the storage that \( o_1 \) occupied, and
8.2 — \( o_1 \) and \( o_2 \) are of the same type (ignoring the top-level cv-qualifiers), and
8.3 — \( o_1 \) is not a complete const object, and
8.4 — neither \( o_1 \) nor \( o_2 \) is a potentially-overlapping subobject (6.7.2), and
8.5 — 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:

```cpp
struct C {
  int i;
  void f();
  const C& operator=(const C&); // lifetime of *this ends
};

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;
}
```

9 If a program ends the lifetime of an object of type \( T \) with static (6.7.5.1), thread (6.7.5.2), or automatic (6.7.5.3) storage duration and if \( T \) has a non-trivial destructor, the program must ensure that an object of the original type occupies the same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [Example:

```cpp
class T { };
```

30) 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. [Example:

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

const B b;

void h() {
  b.~B();
  new (const_cast<B*>(&b)) const B; // undefined behavior
}
```
— end example]

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

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: 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 (6.8.1) 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 (7.3.8, 7.6.1.3, 7.6.1.8, 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:
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

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:

(1.1) — static storage duration
(1.2) — thread storage duration
(1.3) — automatic storage duration
(1.4) — 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 (7.6.2.7).

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 (6.8.2). 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.\(^{31}\)

6.7.5.1 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 11.10.5.

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

6.7.5.2 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: 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]

6.7.5.3 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: 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 11.10.5.

\(^{31}\) Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.
6.7.5.4 Dynamic storage duration

Objects can be created dynamically during program execution (6.9.1), using new-expressions (7.6.2.7), and destroyed using delete-expressions (7.6.2.8). 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: The non-allocating forms described in 17.6.2.3 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 (17.6.2). 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 (16.5.4.6). 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: 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> (17.6.1) 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).

If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in 6.7.5.4.1 and 6.7.5.4.2, the behavior is undefined.

6.7.5.4.1 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 (9.3.3.6). 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 (6.8.2) 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 17.6.2.1 and 17.6.2.2, 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.\(^{32}\)

\(^{32}\) The intent is to have operator new() implementable by calling std::malloc() or std::calloc(), so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.
For an allocation function other than a reserved placement allocation function (17.6.2.3), the pointer returned on a successful call shall represent the address of storage that is aligned as follows:

- If the allocation function takes an argument of type `std::align_val_t`, the storage will have the alignment specified by the value of this argument.
- Otherwise, if the allocation function is named `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.
- Otherwise, the storage is aligned for any object that does not have new-extended alignment and is of the requested size.

An allocation function that fails to allocate storage can invoke the currently installed new-handler function (17.6.3.3), if any. [Note: A program-supplied allocation function can obtain the address of the currently installed new_handler using the `std::get_new_handler` function (17.6.3.5). — end note] 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 `std::bad_alloc` (17.6.3.1).

A global allocation function is only called as the result of a new expression (7.6.2.7), or called directly using the function call syntax (7.6.1.2), 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. [Note: In particular, a global allocation function is not called to allocate storage for objects with static storage duration (6.7.5.1), for objects or references with thread storage duration (6.7.5.2), for objects of type `std::type_info` (7.6.1.7), or for an exception object (14.2). — end note]

### 6.7.5.4.2 Deallocation functions
[basic.stc.dynamic.deallocation]

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 **destroying operator delete** if it has at least two parameters and its second parameter is of type `std::destroying_delete_t`. A destroying operator delete shall be a class member function named `operator delete`. [Note: Array deletion cannot use a destroying operator delete. — end note]

Each deallocation function shall return `void`. If the function is a destroying operator delete declared in class type C, the type of its first parameter shall be `C*`; otherwise, the type of its first parameter shall be `void*`. A deallocation function may have more than one parameter. A **usual deallocation function** is a deallocation function whose parameters after the first are

- optionally, a parameter of type `std::destroying_delete_t`, then
- optionally, a parameter of type `std::size_t`, then
- optionally, a parameter of type `std::align_val_t`.

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 (6.8.2), the deallocation function shall deallocate the storage referenced by the pointer, ending the duration of the region of storage.

### 6.7.5.4.3 Safely-derived pointers
[basic.stc.dynamic.safety]

A **traceable pointer object** is

- an object of an object pointer type (6.8.2), or
- an object of an integral type that is at least as large as `std::intptr_t`, or

33) The global `operator delete(void*, std::size_t)` precludes use of an allocation function `void operator new(std::size_t, std::size_t)` as a placement allocation function (C.3.2).
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 (7.3.11, 7.6.1.3, 7.6.1.8, 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.

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, if that result converted by `reinterpret_cast<void*>` would compare equal to a safely-derived pointer computable from `reinterpret_cast<void*>(P)`.

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: 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.

### 6.7.5.5 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 (6.8.1, 6.8.2) 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 might 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:

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

34) This subclause does not impose restrictions on indirection through pointers to memory not allocated by `::operator new`. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because indirection through pointers to memory allocated by `std::malloc` is not restricted.
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 \texttt{alignof} operator reflects the alignment requirement of the type in the complete-object case.

An extended alignment is represented by an alignment greater than \texttt{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: 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 \texttt{__STDCPP_DEFAULT_NEW_ALIGNMENT__} (15.11).

Alignments are represented as values of the type \texttt{std::size_t}. Valid alignments include only those values returned by an \texttt{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 \texttt{alignof} expression (7.6.2.5). Furthermore, the narrow character types (6.8.1) shall have the weakest alignment requirement. [Note: 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: 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 (20.15.7.6) 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 (7.3.4),
- 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: 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: 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 \texttt{decltype-specifier} (7.6.1.2), 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. [Note: Temporary objects are materialized:

- when binding a reference to a prvalue (9.4.3, 7.6.1.3, 7.6.1.6, 7.6.1.8, 7.6.1.10, 7.6.3),
- when performing member access on a class prvalue (7.6.1.4, 7.6.4),
- when performing an array-to-pointer conversion or subscripting on an array prvalue (7.3.2, 7.6.1.1),
- when initializing an object of type \texttt{std::initializer_list<T>} from a \texttt{braced-init-list} (9.4.4),
- for certain unevaluated operands (7.6.1.7, 7.6.2.4), and
- when a prvalue that has type other than \texttt{cv void} appears as a discarded-value expression (7.2).
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

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 (11.4.3), 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: This latitude is granted to allow objects of class type to be passed to or returned from functions in registers. —end note]

When an implementation introduces a temporary object of a class that has a non-trivial constructor (11.4.4.1, 11.4.4.2), 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 (11.4.6). 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.

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 (7.5.5.2, 11.4.4.2). In either case, if the constructor has one or more default arguments, 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 gvalue to which the reference is bound was obtained through one of the following:

(6.1) — a temporary materialization conversion (7.3.4),
(6.2) — ( expression ), where expression is one of these expressions,

The same rules apply to initialization of an initializer_list object (9.4.4) with its underlying temporary array.
subscripting (7.6.1.1) of an array operand, where that operand is one of these expressions,

— a class member access (7.6.1.4) 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 (7.6.1.10),
— static_cast (7.6.1.8),
— dynamic_cast (7.6.1.6), or
— reinterpret_cast (7.6.1.9)

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:

```cpp
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: An explicit type conversion (7.6.1.3, 7.6.3) is interpreted as a sequence of elementary casts, covered above.]
```

— end example]

— A temporary object bound to a reference parameter in a function call (7.6.1.2) 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 (8.7.3) 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 (7.6.2.7) persists until the completion of the full-expression containing the new-initializer. [Note: This may introduce a dangling reference.]

— end note]

The exceptions to this lifetime rule are:

— A temporary object bound to a reference parameter in a function call (7.6.1.2) 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 (8.7.3) 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 (7.6.2.7) persists until the completion of the full-expression containing the new-initializer. [Note: This may introduce a dangling reference.]

— end note]

— end example]
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 (6.7.5.1, 6.7.5.2, 6.7.5.3); that is, if \texttt{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 \texttt{obj1} is destroyed; if \texttt{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 \texttt{obj2} is destroyed.

8 \[Example:
```cpp
struct S {
    S();
    S(int);
    friend S operator+(const S&, const S&);
    ~S();
};
S obj1;
const S& cr = S(16)+S(23);
S obj2;
The expression \texttt{S(16) + S(23)} creates three temporaries: a first temporary \texttt{T1} to hold the result of the expression \texttt{S(16)}, a second temporary \texttt{T2} to hold the result of the expression \texttt{S(23)}, and a third temporary \texttt{T3} to hold the result of the addition of these two expressions. The temporary \texttt{T3} is then bound to the reference \texttt{cr}. It is unspecified whether \texttt{T1} or \texttt{T2} is created first. On an implementation where \texttt{T1} is created before \texttt{T2}, \texttt{T2} shall be destroyed before \texttt{T1}. The temporaries \texttt{T1} and \texttt{T2} are bound to the reference parameters of \texttt{operator+}; these temporaries are destroyed at the end of the full-expression containing the call to \texttt{operator+}. The temporary \texttt{T3} bound to the reference \texttt{cr} is destroyed at the end of \texttt{cr}'s lifetime, that is, at the end of the program. In addition, the order in which \texttt{T3} is destroyed takes into account the destruction order of other objects with static storage duration. That is, because \texttt{obj1} is constructed before \texttt{T3}, and \texttt{T3} is constructed before \texttt{obj2}, \texttt{obj2} shall be destroyed before \texttt{T3}, and \texttt{T3} shall be destroyed before \texttt{obj1}. —end example\]

6.8 Types \[basic.types\]

1 \[Note: 6.8 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (6.7.2), references (9.3.3.2), or functions (9.3.3.5). —end note\]

2 For any object (other than a potentially-overlapping subobject) of trivially copyable type \texttt{T}, whether or not the object holds a valid value of type \texttt{T}, the underlying bytes (6.7.1) making up the object can be copied into an array of \texttt{char}, \texttt{unsigned char}, or \texttt{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. \[Example:
```cpp
constexpr std::size_t N = sizeof(T);
char buf[N];
T obj; // obj initialized to its original value
std::memcpy(buf, &obj, N); // between these two calls to std::memcpy, obj might be modified
std::memcpy(&obj, buf, N); // at this point, each subobject of obj of scalar type holds its original value
—end example\]

3 For any trivially copyable type \texttt{T}, if two pointers to \texttt{T} point to distinct \texttt{T} objects \texttt{obj1} and \texttt{obj2}, where neither \texttt{obj1} nor \texttt{obj2} is a potentially-overlapping subobject, if the underlying bytes (6.7.1) making up \texttt{obj1} are copied into \texttt{obj2}, \texttt{obj2} shall subsequently hold the same value as \texttt{obj1}. \[Example:
```cpp
T* t1p;
T* t2p;
// provided that t2p points to an initialized object ...
std::memcpy(t1p, t2p, sizeof(T));
// at this point, every subobject of trivially copyable type in *t1p contains
// the same value as the corresponding subobject in *t2p
—end example\]

4 The object representation of an object of type \texttt{T} is the sequence of \texttt{N unsigned char} objects taken up by the object of type \texttt{T}, where \texttt{N} equals \texttt{sizeof(T)}. The value representation of an object of type \texttt{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.\footnote{38}

5 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.\footnote{39} Incompletely-defined object types and cv void are incomplete types (6.8.1). Objects shall not be defined to have an incomplete type.

6 A class type (such as “class X”) might 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 might 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 might 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:

```c

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

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]  

7 [Note: The rules for declarations and expressions describe in which contexts incomplete types are prohibited. — end note]  

8 An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void.

9 Arithmetic types (6.8.1), enumeration types, pointer types, pointer-to-member types (6.8.2), std::nullptr_t, and cv-qualified (6.8.3) 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.

10 A type is a literal type if it is:

(10.1) — cv void; or

\(\text{§ 6.8} \)
— a scalar type; or
— a reference type; or
— an array of literal type; or
— a possibly cv-qualified class type (Clause 11) that has all of the following properties:
  — it has a constexpr destructor (9.2.5),
  — it is either a closure type (7.5.5.1), an aggregate type (9.4.1), 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,
  — if it is a union, at least one of its non-static data members is of non-volatile literal type, and
  — if it is not a union, all of its non-static data members and base classes are of non-volatile literal types.

[Note: 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).

6.8.1 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: 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: 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. [Example: 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>Type</th>
<th>Minimum width $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>16</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>64</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: Padding bits have unspecified value, but cannot cause

40) This is also known as two’s complement representation.
traps. In contrast, see ISO C 6.2.6.2. [Note: 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: This requirement does not hold for other types. — end note] [Note: 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> (17.4.1).

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: 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: Enumerations (9.7.1) are not integral; however, unscoped enumerations can be promoted to integral types as specified in 7.3.6. — 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: 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 (7.6.1.3, 7.6.1.8, 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 a second or third operand of ?: (7.6.16), as the operand of typeid, noexcept, or decltype, as the expression in a return statement (8.7.3) 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 (7.3.11). Such values participate in the pointer and the pointer-to-member conversions (7.3.11, 7.3.12). sizeof(std::nullptr_t) shall be equal to sizeof(void*).

The types described in this subclause are called fundamental types. [Note: Even if the implementation defines two or more fundamental types to have the same value representation, they are nevertheless different types. — end note]
6.8.2 Compound types

1 Compound types can be constructed in the following ways:

(1.1) — arrays of objects of a given type, 9.3.3.4;

(1.2) — functions, which have parameters of given types and return \texttt{void} or references or objects of a given type, 9.3.3.5;

(1.3) — pointers to \texttt{cv void} or objects or functions (including static members of classes) of a given type, 9.3.3.1;

(1.4) — references to objects or functions of a given type, 9.3.3.2. There are two types of references:

(1.4.1) — value reference

(1.4.2) — rvalue reference

(1.5) — classes containing a sequence of objects of various types (Clause 11), a set of types, enumerations and functions for manipulating these objects (11.4.1), and a set of restrictions on the access to these entities (11.9);

(1.6) — unions, which are classes capable of containing objects of different types at different times, 11.5;

(1.7) — enumerations, which comprise a set of named constant values. Each distinct enumeration constitutes a different enumerated type, 9.7.1;

(1.8) — pointers to non-static class members, which identify members of a given type within objects of a given class, 9.3.3.3. Pointers to data members and pointers to member functions are collectively called \texttt{pointer-to-member} types.

2 These methods of constructing types can be applied recursively; restrictions are mentioned in 9.3.3. Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type \texttt{std::size_t} (17.2) is ill-formed.

3 The type of a pointer to \texttt{cv void} or a pointer to an object type is called an \texttt{object pointer type}. [Note: A pointer to \texttt{void} does not have a pointer-to-object type, however, because \texttt{void} is not an object type. — end note] The type of a pointer that can designate a function is called a \texttt{function pointer type}. A pointer to an object of type \texttt{T} is referred to as a “pointer to \texttt{T}”. [Example: A pointer to an object of type \texttt{int} is referred to as “pointer to \texttt{int}” and a pointer to an object of class \texttt{X} is called a “pointer to \texttt{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:

(3.1) — a pointer to an object or function (the pointer is said to \textit{point} to the object or function), or

(3.2) — a pointer past the end of an object (7.6.6), or

(3.3) — the \texttt{null pointer value} for that type, or

(3.4) — an invalid pointer value.

A value of a pointer type that is a pointer to or past the end of an object \textit{represents the address} of the first byte in memory (6.7.1) occupied by the object\footnote{Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.} or the first byte in memory after the end of the storage occupied by the object, respectively. [Note: 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 \texttt{x} of \texttt{n} elements is considered to be equivalent to a pointer to a hypothetical array element \texttt{n} of \texttt{x} and an object of type \texttt{T} that is not an array element is considered to belong to an array with one element of type \texttt{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: 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]

4 Two objects \texttt{a} and \texttt{b} are \texttt{pointer-interconvertible} if:

(4.1) — they are the same object, or

(4.2) — one is a union object and the other is a non-static data member of that object (11.5), or

\footnote{For an object that is not within its lifetime, this is the first byte in memory that it will occupy or used to occupy.}
— 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

— 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 (7.6.1.9). [Note: An array object and its first element are not pointer-interconvertible, even though they have the same address. — end note]

5 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*.

6.8.3 CV-qualifiers [basic.type.qualifier]

1 A type mentioned in 6.8.1 and 6.8.2 is a cv-unqualified type. Each type which is a cv-unqualified complete or incomplete 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 cv-qualifiers specified in the decl-specifier-seq (9.2), declarator (9.3), type-id (9.3.1), or new-type-id (7.6.2.7) 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). 43

2 Except for array types, a compound type (6.8.2) 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: 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 (9.3.3.4). — end note]

[Example: typedef char CA[5]; typedef const char CC; CC arr1[5] = { 0 }; const CA arr2 = { 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: See 9.3.3.5 and 11.4.2.1 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 [tab:basic.type.qualifier.rel]

<table>
<thead>
<tr>
<th>no cv-qualifier</th>
<th>const</th>
</tr>
</thead>
<tbody>
<tr>
<td>no cv-qualifier</td>
<td>volatile</td>
</tr>
<tr>
<td>no cv-qualifier</td>
<td>const volatile</td>
</tr>
<tr>
<td>const</td>
<td>const volatile</td>
</tr>
<tr>
<td>volatile</td>
<td>const volatile</td>
</tr>
</tbody>
</table>

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: The type corresponding to the

43) 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.
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

6.8.4 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.

(1.5) — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

(1.6) — The rank of char shall equal the rank of signed char and unsigned char.

(1.7) — The rank of bool shall be less than the rank of all other standard integer types.

(1.8) — The ranks of char8_t, char16_t, char32_t, and wchar_t shall equal the ranks of their underlying types (6.8.1).

(1.9) — 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.

(1.10) — 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: The integer conversion rank is used in the definition of the integral promotions (7.3.6) and the usual arithmetic conversions (7.4). — end note]

6.9 Program execution [basic.exec]

6.9.1 Sequential execution [intro.execution]

1 An instance of each object with automatic storage duration (6.7.5.3) 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 (7.6.2.3), or receipt of a signal).

2 A constituent expression is defined as follows:

(2.1) — The constituent expression of an expression is that expression.

(2.2) — 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.

(2.3) — The constituent expressions of a brace-or-equal-initializer of the form = initializer-clause are the constituent expressions of the initializer-clause.

[Example:

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]

3 The immediate subexpressions of an expression E are

(3.1) — the constituent expressions of E’s operands (7.2),

(3.2) — any function call that E implicitly invokes,

(3.3) — if E is a lambda-expression (7.5.5), the initialization of the entities captured by copy and the constituent expressions of the initializer of the init-captures,
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:]
```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
```

[Note: 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 (9.3.3.6) 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.

8 **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: 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: 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.

9 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.44

10 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [Note: 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: The next subclause imposes similar, but more complex restrictions on potentially concurrent computations. —end note]

[Example:

```c
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
}
```

—end example]

11 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.45 [Note: 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: Evaluation of a new-expression invokes one or more allocation and constructor functions; see 7.6.2.7. For another example, invocation of a conversion function (11.4.7.2) 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, whatever the syntax of the expression that calls the function might be.

12 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: 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]

———

44) 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.

45) In other words, function executions do not interleave with each other.
6.9.2 Multi-threaded executions and data races  
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: 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. Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each 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: 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.

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.

6.9.2.1 Data races
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: In some cases, there may 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] 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.

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: 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]

All modifications to a particular atomic object M occur in some particular total order, called the modification order of M. [Note: 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 may observe modifications to different objects in inconsistent orders. — end note] 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.

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: 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] 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]

The value of A is used as an operand of B, unless:

- B is an invocation of any specialization of std::kill_dependency (31.4), or

An evaluation A carries a dependency to an evaluation B if

---

[46] 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 (6.8.2).
A is the left operand of a built-in logical AND (&&, see 7.6.14) or logical OR (|, see 7.6.15) operator, or

A is the left operand of a conditional (?; see 7.6.16) operator, or

A is the left operand of the built-in comma (,) operator (7.6.20); or

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

for some evaluation X, A carries a dependency to X, and X carries a dependency to B.

[Note: “Carries a dependency to” is a subset of “is sequenced before”, and is similarly strictly intra-thread. —end note]

An evaluation A is dependency-ordered before an evaluation B if

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

for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[Note: The relation “is dependency-ordered before” is analogous to “synchronizes with”, but uses release/consume in place of release/acquire. —end note]

An evaluation A inter-thread happens before an evaluation B if

A synchronizes with B, or

A is dependency-ordered before B, or

for some evaluation X

A synchronizes with X and X is sequenced before B, or

A is sequenced before X and X inter-thread happens before B, or

A inter-thread happens before X and X inter-thread happens before B.

[Note: 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]

An evaluation A happens before an evaluation B (or, equivalently, B happens after A) if:

A is sequenced before B, or

A inter-thread happens before B.

The implementation shall ensure that no program execution demonstrates a cycle in the “happens before” relation. [Note: This cycle would otherwise be possible only through the use of consume operations. —end note]

An evaluation A simply happens before an evaluation B if either

A is sequenced before B, or

A synchronizes with B, or

A simply happens before X and X simply happens before B.

[Note: In the absence of consume operations, the happens before and simply happens before relations are identical. —end note]

An evaluation A strongly happens before an evaluation D if, either

A is sequenced before D, or
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:

1. A happens before B and
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: 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: 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]

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: This requirement is known as write-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: This requirement is known as read-write coherence. — end note]

Two actions are potentially concurrent if

1. they are performed by different threads, or
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.
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 must perform an undefined operation. —end note

22 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 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.

23 [Note: 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 may alias is also generally precluded, since this may violate the coherence rules. —end note]

24 [Note: Transformations that introduce a speculative read of a potentially shared memory location may 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]

6.9.2.2 Forward progress

1 The implementation may assume that any thread will eventually do one of the following:

- (1.1) terminate,
- (1.2) make a call to a library I/O function,
- (1.3) perform an access through a volatile glvalue, or
- (1.4) perform a synchronization operation or an atomic operation.

[Note: This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. —end note]

2 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.

- (2.1) If there is only one thread that is not blocked (3.6) in a standard library function, a lock-free execution in that thread shall complete. [Note: Concurrently executing threads may 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]
- (2.2) When one or more lock-free executions run concurrently, at least one should complete. [Note: It is difficult for some implementations to provide absolute guarantees to this effect, since repeated and particularly inopportune interference from other threads may prevent forward progress, e.g., by repeatedly stealing a cache line for unrelated purposes between load-locked and store-conditional instructions. Implementations should ensure that such effects cannot indefinitely delay progress under expected operating conditions, and that such anomalies can therefore safely be ignored by programmers. Outside this document, this property is sometimes termed lock-free. —end note]

3 During the execution of a thread of execution, each of the following is termed an execution step:

- (3.1) termination of the thread of execution,
- (3.2) performing an access through a volatile glvalue, or
- (3.3) completion of a call to a library I/O function, a synchronization operation, or an atomic operation.

4 An invocation of a standard library function that blocks (3.6) is considered to continuously execute execution steps while waiting for the condition that it blocks on to be satisfied. [Example: 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]
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. [Note: This is required regardless of whether or not other threads of executions (if any) have been or are making progress. To eventually fulfill this requirement means that this will happen in an unspecified but finite amount of time. —end note]

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 (32.4.2) or std::jthread (32.4.3) provide concurrent forward progress guarantees. [Note: General-purpose implementations should provide these guarantees. —end note]

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.

[Note: This does not specify a requirement for when to start this thread of execution, which will typically be specified by the entity that creates this thread of execution. For example, a thread of execution that provides concurrent forward progress guarantees and executes tasks from a set of tasks in an arbitrary order, one after the other, satisfies the requirements of parallel forward progress for these tasks. —end note]

For a thread of execution providing weakly parallel forward progress guarantees, the implementation does not ensure that the thread will eventually make progress.

[Note: Threads of execution providing weakly parallel forward progress guarantees cannot be expected to make progress regardless of whether other threads make progress or not; however, blocking with forward progress guarantee delegation, as defined below, can be used to ensure that such threads of execution make progress eventually. —end note]

Concurrent forward progress guarantees are stronger than parallel forward progress guarantees, which in turn are stronger than weakly parallel forward progress guarantees. [Note: For example, some kinds of synchronization between threads of execution may only make progress if the respective threads of execution provide parallel forward progress guarantees, but will fail to make progress under weakly parallel guarantees. —end note]

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. [Note: It is unspecified which thread or threads of execution in S are chosen and for which number of execution steps. The strengthening is not permanent and not necessarily in place for the rest of the lifetime of the affected thread of execution. As long as P is blocked, the implementation has to eventually select and potentially strengthen a thread of execution in S. —end note]. Once a thread of execution in S terminates, it is removed from S. Once S is empty, P is unblocked.

[Note: A thread of execution B thus can temporarily provide an effectively stronger forward progress guarantee for a certain amount of time, due to a second thread of execution A being blocked on it with forward progress guarantee delegation. In turn, if B then blocks with forward progress guarantee delegation on C, this may also temporarily provide a stronger forward progress guarantee to C. —end note]

[Note: If all threads of execution in S finish executing (e.g., they terminate and do not use blocking synchronization incorrectly), then P’s execution of the operation that blocks with forward progress guarantee delegation will not result in P’s progress guarantee being effectively weakened. —end note]

[Note: This does not remove any constraints regarding blocking synchronization for threads of execution providing parallel or weakly parallel forward progress guarantees because the implementation is not required to strengthen a particular thread of execution whose too-weak progress guarantee is preventing overall progress. —end note]

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

1 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, and in which variables of static storage duration might be initialized (6.9.3.2) and destroyed (6.9.3.4). It is implementation-defined whether a program in a freestanding environment is required to define a main function. [Note: In a freestanding environment, startup and termination is implementation-defined; startup contains the execution of constructors for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration. —end note]

2 An implementation shall not predefine 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

(2.1) — a function of () returning int and
(2.2) — a function of (int, pointer to pointer to char) returning int

as the type of main (9.3.3.5). In the latter form, for purposes of exposition, the first function parameter is called argv and the second function parameter is called argv, where argv shall be the number of arguments passed to the program from the environment in which the program is run. If argv 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 (NTMBSs) (16.4.2.5.2) 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 argv shall be non-negative. The value of argv[argc] shall be 0. [Note: It is recommended that any further (optional) parameters be added after argv. —end note]

3 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: Member functions, classes, and enumerations can be called main, as can entities in other namespaces. —end example]

4 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 (11.4.6). 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.

5 A return statement (8.7.3) 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

1 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.

2 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 (6.7.5.1) or thread storage duration (6.7.5.2) 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 (6.9.2.1) any dynamic initialization. [Note: 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]

3 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

(3.1) — 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: As a consequence, if the initialization of an object \texttt{obj1} refers to an object \texttt{obj2} of namespace scope potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified whether the value of \texttt{obj2} used will be the value of the fully initialized \texttt{obj2} (because \texttt{obj2} was statically initialized) or will be the value of \texttt{obj2} merely zero-initialized. For example,

```c
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1; // unspecified:
    // may be statically initialized to 0.0 or
    // dynamically initialized to 0.0 if \texttt{d1} is
    // dynamically initialized, or 1.0 otherwise
double d1 = fd(); // may be initialized statically or dynamically to 1.0
```
—end note]

6.9.3.3 Dynamic initialization of non-local variables

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: An explicitly specialized non-inline static data member or variable template specialization has ordered initialization. —end note]

A declaration \(D\) is appearance-ordered before a declaration \(E\) if

1. \(D\) appears in the same translation unit as \(E\), or
2. the translation unit containing \(E\) has an interface dependency on the translation unit containing \(D\), in either case prior to \(E\).

Dynamic initialization of non-local variables \(V\) and \(W\) with static storage duration are ordered as follows:

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
2. 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. otherwise, the initialization of \(V\) strongly happens before the initialization of \(W\).
4. 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.
5. Otherwise, the initializations of \(V\) and \(W\) are indeterminately sequenced.

[Note: This definition permits initialization of a sequence of ordered variables concurrently with another sequence. —end note]

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.

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 \texttt{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.\(^{47}\) It is implementation-defined in which threads and at which points in the program such deferred dynamic initialization occurs. [Note: Such points should be chosen in a way that allows the programmer to avoid deadlocks. —end note] [Example:
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 (14.6.1).

6.9.3.4 Termination [basic.start.term]

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: 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 (14.6.1).
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.

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: 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: Clause 7 defines the syntax, order of evaluation, and meaning of expressions. 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: 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 ++a meaning 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 12.4.1.2, 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: 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: The implementation may regroup operators according to the usual mathematical rules only where the operators really are associative or commutative. For example, in the following fragment

```c
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```c
a = ((a + 32760) + b) + 5;
```

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 int is [-32768, +32767], the implementation cannot rewrite this expression as

```c
a = ((a + b) + 32765);
```

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

```c
a = ((a + 32765) + b);
```
or

```c
a = (a + (b + 32765));
```

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.]

---

48) The precedence of operators is not directly specified, but it can be derived from the syntax.
49) Overloaded operators are never assumed to be associative or commutative.
50) The cast and assignment operators must still perform their specific conversions as described in 7.6.1.3, 7.6.3, 7.6.1.8 and 7.6.19.
7.2 Properties of expressions

7.2.1 Value category

Expressions are categorized according to the taxonomy in Figure 1.

Figure 1: Expression category taxonomy

(1.1) A glvalue is an expression whose evaluation determines the identity of an object, bit-field, or function.
(1.2) A prvalue is an expression whose evaluation initializes an object or a bit-field, 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.
(1.3) An xvalue is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime).
(1.4) An lvalue is a glvalue that is not an xvalue.
(1.5) 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: 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: 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: An expression is an xvalue if it is:

(4.1) the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type (7.6.1.2),
(4.2) a cast to an rvalue reference to object type (7.6.1.3, 7.6.1.6, 7.6.1.8, 7.6.1.9, 7.6.1.10, 7.6.3),
(4.3) a subscripting operation with an xvalue array operand (7.6.1.1),
(4.4) a class member access expression designating a non-static data member of non-reference type in which the object expression is an xvalue (7.6.1.4), or
(4.5) 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:

```c
struct A {
    int m;
};
A&& operator+(A, A);
A&& f();
```]
A a;
A&& ar = static_cast<A&&>(a);

The expressions f().m, static_cast<A&&>(a), and a + a are xvalues. The expression ar is an lvalue.

—end example[

5 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 is used to compute the value of an operand of a built-in operator or a prvalue that has type cv void has no result object. [Note: 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]

6 Whenever a glvalue appears as an operand of an operator that expects a prvalue for that operand, the lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), or function-to-pointer (7.3.3) standard conversions are applied to convert the expression to a prvalue. [Note: An attempt to bind a rvalue reference to an lvalue is not such a context; see 9.4.3. —end note] [Note: Because cv-qualifiers are removed from the type of an expression of non-class type when the expression is converted to a prvalue, an lvalue of type const int can, for example, be used where a prvalue of type int is required. —end note] [Note: There are no prvalue bit-fields; if a bit-field is converted to a prvalue (7.3.1), a prvalue of the type of the bit-field is created, which might then be promoted (7.3.6). —end note]

7 Whenever a prvalue appears as an operand of an operator that expects a glvalue for that operand, the temporary materialization conversion (7.3.4) is applied to convert the expression to an xvalue.

8 The discussion of reference initialization in 9.4.3 and of temporaries in 6.7.7 indicates the behavior of lvalues and rvalues in other significant contexts.

9 Unless otherwise indicated (9.2.8.2), 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 (11.7.3). A glvalue shall not have type cv void. [Note: A glvalue may 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]

10 An lvalue is modifiable unless its type is const-qualified or is a function type. [Note: A program that attempts to modify an object through a nonmodifiable lvalue or through an rvalue is ill-formed (7.6.19, 7.6.1.5, 7.6.2.2). —end note]

11 If a program attempts to access (3.1) the stored value of an object through a glvalue whose type is not similar (7.3.5) to one of the following types the behavior is undefined:51

(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: Unlike in C, C++ has no accesses of class type. —end note]

7.2.2 Type [expr.type]

1 If an expression initially has the type “reference to T” (9.3.3.2, 9.4.3), 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: Before the lifetime of the reference has started or after it has ended, the behavior is undefined (see 6.7.3). —end note]

2 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.

3 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;
(3.2) — if either p1 or p2 is a null pointer constant, T2 or T1, respectively;

51) The intent of this list is to specify those circumstances in which an object may or may not be aliased.
(3.3) if \( T_1 \) or \( T_2 \) is “pointer to \( cv1 \) void” and the other type is “pointer to \( cv2 \) \( T \)”, where \( T \) is an object type or void, “pointer to \( cv12 \) void”, where \( cv12 \) is the union of \( cv1 \) and \( cv2 \);

(3.4) if \( T_1 \) or \( T_2 \) is “pointer to noexcept function” and the other type is “pointer to function”, where the function types are otherwise the same, “pointer to function”;

(3.5) if \( T_1 \) is “pointer to \( cv1 \) \( C_1 \)” and \( T_2 \) is “pointer to \( cv2 \) \( C_2 \)”, where \( C_1 \) is reference-related to \( C_2 \) or \( C_2 \) is reference-related to \( C_1 \) (9.4.3), the cv-combined type (7.3.5) of \( T_1 \) and \( T_2 \) or the cv-combined type of \( T_2 \) and \( T_1 \), respectively;

(3.6) if \( T_1 \) or \( T_2 \) is “pointer to member of \( C_1 \) of type function”, the other type is “pointer to member of \( C_2 \) of type noexcept function”, and \( C_1 \) is reference-related to \( C_2 \) or \( C_2 \) is reference-related to \( C_1 \) (9.4.3), where the function types are otherwise the same, “pointer to member of \( C_2 \) of type function” or “pointer to member of \( C_1 \) of type function”, respectively;

(3.7) if \( T_1 \) is “pointer to member of \( C_1 \) of type \( cv1 \) \( U \)” and \( T_2 \) is “pointer to member of \( C_2 \) of type \( cv2 \) \( U \)”, for some non-function type \( U \), where \( C_1 \) is reference-related to \( C_2 \) or \( C_2 \) is reference-related to \( C_1 \) (9.4.3), the cv-combined type of \( T_2 \) and \( T_1 \) or the cv-combined type of \( T_1 \) and \( T_2 \), respectively;

(3.8) if \( T_1 \) and \( T_2 \) are similar types (7.3.5), the cv-combined type of \( T_1 \) and \( T_2 \);

(3.9) otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.

[Example:  
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 const void”; the composite pointer type of \( pi \) and \( pci \) is “pointer to const pointer to const int”. — end example]

7.2.3 Context dependence [expr.context]

1 In some contexts, unevaluated operands appear (7.5.7, 7.6.1.7, 7.6.2.4, 7.6.2.6, 9.2.8.2, 13.1, 13.7.8). An unevaluated operand is not evaluated. [Note: In an unevaluated operand, a non-static class member may 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]

2 In some contexts, an expression only appears for its side effects. Such an expression is called a discarded-value expression. The array-to-pointer (7.3.2) and function-to-pointer (7.3.3) standard conversions are not applied. The lvalue-to-rvalue conversion (7.3.1) is applied if and only if the expression is a glvalue of volatile-qualified type and it is one of the following:

(2.1) ( expression ), where expression is one of these expressions,

(2.2) id-expression (7.5.4),

(2.3) subscripting (7.6.1.1),

(2.4) class member access (7.6.1.4),

(2.5) indirection (7.6.2.1),

(2.6) pointer-to-member operation (7.6.4),

(2.7) conditional expression (7.6.16) where both the second and the third operands are one of these expressions, or

(2.8) comma expression (7.6.20) where the right operand is one of these expressions.

[Note: 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 (7.3.4) is applied. [Note: 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 [conv]

1 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:
(1.1) — Zero or one conversion from the following set: lvalue-to-rvalue conversion, array-to-pointer conversion, and function-to-pointer conversion.

(1.2) — 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.

(1.3) — Zero or one function pointer conversion.

(1.4) — Zero or one qualification conversion.

[Note: 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: Expressions with a given type will be implicitly converted to other types in several contexts:

(2.1) — When used as operands of operators. The operator’s requirements for its operands dictate the destination type (7.6).

(2.2) — When used in the condition of an if statement (8.5.1) or iteration statement (8.6). The destination type is bool.

(2.3) — When used in the expression of a switch statement (8.5.2). The destination type is integral.

(2.4) — 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, 9.4.3.

— end note]

3 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 a 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 \( cv \ T \) or reference to \( cv \ T \) such that \( T \) is allowed by the context. There shall be exactly one such \( T \).

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 (9.3.3.2), an xvalue if \( T \) is an rvalue reference to object type, and a prvalue otherwise. The expression \( E \) is used as a glvalue if and only if the initialization uses it as a glvalue.

7 [Note: For class types, user-defined conversions are considered as well; see 11.4.7. In general, an implicit conversion sequence (12.4.3.1) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. — end note]

8 [Note: 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]

7.3.1 Lvalue-to-rvalue conversion [conv.lval]

1 A glvalue (7.2.1) of a non-function, non-array type \( T \) can be converted to a prvalue.\(^{52}\) 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 is the \( cv \)-unqualified version of \( T \). Otherwise, the type of the prvalue is \( T \).\(^{53}\)

2 When an lvalue-to-rvalue conversion is applied to an expression \( E \), and either

(2.1) — \( E \) is not potentially evaluated, or

\(^{52}\) 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.

\(^{53}\) 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.
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:

```c
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
```

— end example]

The result of the conversion is determined according to the following rules:

(3.1) — If \( T \) is `std::nullptr_t`, the result is a null pointer constant (7.3.11). [Note: 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]

(3.2) — Otherwise, if \( T \) has a class type, the conversion copy-initializes the result object from the glvalue.

(3.3) — Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.7.5.4.2, 6.7.5.4.3), the behavior is implementation-defined.

(3.4) — Otherwise, the object indicated by the glvalue is read (3.1), and the value contained in the object is the prvalue result.

[Note: See also 7.2.1. — end note]

### 7.3.2 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 (7.3.4) is applied. The result is a pointer to the first element of the array.

### 7.3.3 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.

### 7.3.4 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: If \( T \) is a class type (or array thereof), it must have an accessible and non-deleted destructor; see 11.4.6. — end note]

[Example:

```c
struct X { int n; }
int k = X().n;    // OK, X() prvalue is converted to xvalue
```

— end example]

### 7.3.5 Qualification conversions

A cv-decomposition of a type \( T \) is a sequence of \( cv_i \) and \( P_i \) such that \( T \) is

\[
\text{cv}_0 \ P_0 \ \text{cv}_1 \ P_1 \ \cdots \ \text{cv}_{n-1} \ P_{n-1} \ \text{cv}_n \ U \quad \text{for} \ n \geq 0,
\]

where each \( \text{cv}_i \) is a set of cv-qualifiers (6.8.3), and each \( P_i \) is “pointer to” (9.3.3.1), “pointer to member of class \( C_i \) of type” (9.3.3.3), “array of \( N_i \)”, or “array of unknown bound of” (9.3.3.4). If \( P_i \) designates an array, the cv-qualifiers \( \text{cv}_{i+1} \) on the element type are also taken as the cv-qualifiers \( \text{cv}_i \) of the array. [Example: The type denoted by the type-id `const int **` has three cv-decompositions, taking \( U \) as “int”, as “pointer to const int”, and as “pointer to pointer to const int”. — end example] The n-tuple of cv-qualifiers

54) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.
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 unknown bound of” 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:

(3.1) — for every $i > 0$, $cv_i$ is the union of $cv_1$ and $cv_2$;
(3.2) — if either $P_1$ or $P_2$ is “array of unknown bound of”, $P_3$ is “array of unknown bound of”, otherwise it is $P_1$;
(3.3) — if the resulting $cv_i$ is different from $cv_1$ or $cv_2$, or the resulting $P_3$ is different from $P_1$ or $P_2$, then const is added to every $cv_k$ for $0 < k < i$.

where $cv_i$ and $P_i$ are the components of the cv-decomposition of Tj. A prvalue of type T1 can be converted to type T2 if the cv-combined type of T1 and T2 is T2. [Note: If a program could assign a pointer of type T** to a pointer of type 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,]

```c
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 note]  [Note: 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: 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: Function types (including those used in pointer-to-member-function types) are never cv-qualified (9.3.3.5). — end note]

7.3.6 Integral promotions

1 A prvalue of an integer type other than bool, char16_t, char32_t, or wchar_t whose integer conversion rank (6.8.4) 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 (6.8.1) 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 (6.8.4) 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 (11.4.9) 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

§ 7.3.6 92
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*.

### 7.3.7 Floating-point promotion

[conv.fpprom]

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*.

### 7.3.8 Integral conversions

[conv.integral]

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 7.3.14. 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.

### 7.3.9 Floating-point conversions

[conv.double]

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.

### 7.3.10 Floating-integral conversions

[conv.fpint]

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: If the destination type is `bool`, see 7.3.14. — 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: 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.

### 7.3.11 Pointer conversions

[conv.ptr]

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 (6.8.2)
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 (7.3.5). A null pointer constant of
integral type can be converted to a prvalue of type `std::nullptr_t`. *[Note: 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 (6.8.2) 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.
7.3.12 Pointer-to-member conversions

A null pointer constant (7.3.11) 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 (7.3.5).

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 (11.7.1) 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 indirecting 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.

7.3.13 Function pointer conversions

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:

```c
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}
```

7.3.14 Boolean conversions

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

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:

(1.1) — 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.

(1.2) — If either operand is of type long double, the other shall be converted to long double.

(1.3) — Otherwise, if either operand is double, the other shall be converted to double.

(1.4) — Otherwise, if either operand is float, the other shall be converted to float.

(1.5) — Otherwise, the integral promotions (7.3.6) shall be performed on both operands. Then the following rules shall be applied to the promoted operands:

(1.5.1) — If both operands have the same type, no further conversion is needed.

55) The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (7.3.11, 11.7). This inversion is necessary to ensure type safety. Note that a pointer to member is not an object pointer or a function pointer and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a void*.

56) As a consequence, operands of type bool, char8_t, char16_t, char32_t, wchar_t, or an enumerated type are converted to some integral type.
— 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 operand that has unsigned 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 signed integer type can represent all of the values of the type of the operand with unsigned 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.

2 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.1).

7.5 Primary expressions

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 (11.4.2.1) 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: This is because declaration matching does not occur until the complete declarator is known. — end note] [Note: In a trailing-return-type, the class being defined is not required to be complete for purposes of class member access (7.6.1.4). Class members declared later are not visible. [Example:

    struct A {
        char g();
        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 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 this shall not appear in any other context. [Example:

    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, value, 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

An id-expression is a restricted form of a primary-expression. [Note: An id-expression can appear after . and -> operators (7.6.1.4). — end note]

An id-expression that denotes a non-static data member or non-static member function of a class can only be used:

1. as part of a class member access (7.6.1.4) in which the object expression refers to the member’s class or a class derived from that class, or
2. to form a pointer to member (7.6.2.1), or
3. if that id-expression denotes a non-static data member and it appears in an unevaluated operand. [Example:
   struct S {
       int m;
   };
   int i = sizeof(S::m); // OK
   int j = sizeof(S::m + 42); // OK
   — end example]

A potentially-evaluated id-expression that denotes an immediate function (9.2.5) shall appear only

1. as a subexpression of an immediate invocation, or
2. in an immediate function context (7.7).

For an id-expression that denotes an overload set, overload resolution is performed to select a unique function (12.4, 12.5). [Note: 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:
   template<typename T> struct A {
       static void f(int) requires false;
   }
   void g() {
       A<int>::f(); // error: cannot call f
       void (*p1)(int) = A<int>::f; // error: cannot take the address of f
       decltype(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]
7.5.4.1 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: For operator-function-ids, see 12.6; for conversion-function-ids, see 11.4.7.2; 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 7.5.4.3. 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 (11.4.2). —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 (7.5.5.2), the type of the expression is the type of a class member access expression (7.6.1.4) 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: If that lambda-expression is not declared mutable, the type of such an identifier will typically be const qualified. —end note] The type of the expression is the type of the result. [Note: 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 type will be adjusted as described in 7.2.2 if it is cv-qualified or is a reference type. —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:

```c
void f() {
  float x, &r = x;
  [=] {
    decltype(x) y1;  // y1 has type float
    decltype((x)) y2 = y1;  // y2 has type float const & because this lambda
                         // is not mutable and x is an lvalue
    decltype(r) r1 = y1;  // r1 has type float
    decltype((r)) r2 = y2;  // r2 has type float const
  };
}
—end example]
```

7.5.4.2 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 ::

The type denoted by a decltype-specifier in a nested-name-specifier shall be a class or enumeration type.

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; 6.5.3.1 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: 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 (7.5.4.3). 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; 6.5.3.2 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.

7.5.4.3 Destruction [expr.prim.id.dtor]

1 An id-expression that denotes the destructor of a type T names the destructor of T if T is a class type (11.4.6), otherwise the id-expression is said to name a pseudo-destructor.

2 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 (7.6.1.4) that forms the postfix-expression of a function call (7.6.1.2). [Note: Such a call ends the lifetime of the object (7.6.1.2, 6.7.3). —end note]

3 [Example:

```cpp
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)
}
—end example]

7.5.5 Lambda expressions [expr.prim.lambda]

Lambda-expression:

```cpp
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
```

1 A lambda-expression provides a concise way to create a simple function object. [Example:

```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]

2 A lambda-expression is a prvalue whose result object is called the closure object. [Note: A closure object behaves like a function object (20.14). —end note]

3 In the decl-specifier-seq of the lambda-declarator, each decl-specifier shall be one of mutable, constexpr, or consteval. [Note: The trailing requires-clause is described in 9.3. —end note]
4 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 9.2.8.5. [Example:

```c
auto x1 = [](int i) { return i; };  // OK: return type is int
auto x2 = [](int {1, 2};) { return j; };  // error: deducing return type from braced-init-list
auto x3 = []()->auto& { return j; };  // OK: return type is int&
@end example]

A lambda is a generic lambda if the lambda-expression has any generic parameter type placeholders (9.2.8.5), or if the lambda has a template-parameter-list. [Example:

```c
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]

7.5.5.1 Closure types [expr.prim.lambda.closure]

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: This determines the set of namespaces and classes associated with the closure type (6.5.2). 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 (9.4.1). 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,
(2.2) — whether the closure type is trivially copyable (11.2), or
(2.3) — 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) (12.6.3) 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: The function call operator template for a generic lambda might be an abbreviated function template (9.3.3.5). —end note] [Example:

```c
auto glambda = [](auto a, auto&& b) { return a < b; };  // OK
bool b = glambda(3, 3.14);
```

```c
auto vglambda = [](auto printer) {
    return [=](auto&& ... ts) {
        // OK: ts is a function parameter pack
        printer(std::forward<decltype(ts)>(ts)...);
        return [=]() {
            printer(ts ...);
        };
    };
};
```

```c
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 (11.4.2) 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

§ 7.5.5.1
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 (9.2.5). It is an immediate function (9.2.5) if the corresponding lambda-expression’s parameter-declaration-clause is followed by consteval. [Note: Names referenced in the lambda-declarator are looked up in the context in which the lambda-expression appears. — end note] [Example:

auto ID = [](auto a) { return a; }; // OK

struct NonLiteral {
    NonLiteral(int n) : n(n) {} 
    int n;
};
static_assert(ID(NonLiteral(3)).n == 3); // error
— end example]

5 [Example:

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]

6 [Note: The function call operator or operator template may be constrained (13.5.2) by a type-constraint (13.2), a requires-clause (13.1), or a trailing requires-clause (9.3). [Example:

    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]

7 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: 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):

```cpp
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:

```cpp
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
    /* ... */
  }
  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; } 
};
```

—end note]

Example:

```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
```

```cpp
auto glambda = [] (auto a) { return a; };  
f1(glambda);  // OK
f2(glambda);  // error: ID is not convertible
g(glambda);  // error: ambiguous
h(glambda);  // OK: calls #3 since it is convertible from ID
```

(int& (*fpi)(int*)) = [] (auto* a) -> auto& { return *a; };  // OK

—end example]

The value returned by any given specialization of this conversion function template is the address of a
function \( F \) that, when invoked, has the same effect as invoking the generic lambda's corresponding function
call operator template specialization on a default-constructed instance of the closure type. \( F \) is a constexpr
function if the corresponding specialization is a constexpr function and \( F \) is an immediate function if the
function call operator template specialization is an immediate function. [Note: This will result in the implicit
instantiation of the generic lambda's body. The instantiated generic lambda's return type and parameter
types are required to match the return type and parameter types of the pointer to function. — end note]

Example:

```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)
```

—end example]
The conversion function or conversion function template is public, constexpr, non-virtual, non-explicit, const, and has a non-throwing exception specification (14.5).  [Example:

```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).
```cpp
auto NC = [] (auto a) { static int s; return a; };
static_assert(Fwd(NC,3) == 3); // error
```
—end example]

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 (11.4.2.1) and transforming id-expressions referring to non-static class members into class member access expressions using (this) (11.4.2), the compound-statement is considered in the context of the lambda-expression.  [Example:

```cpp
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.

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 (11.4.4.2). It has a deleted copy assignment operator if the lambda-expression has a lambda-capture and defaulted copy and move assignment operators otherwise (11.4.5).  [Note: These special member functions are implicitly defined as usual, and might therefore be defined as deleted. — end note]

The closure type associated with a lambda-expression has an implicitly-declared destructor (11.4.6).

A member of a closure type shall not be explicitly instantiated (13.9.2), explicitly specialized (13.9.3), or named in a friend declaration (11.9.3).

### 7.5.5.2 Captures

[expr.prim.lambda.capture]  

```cpp
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
```
The body of a lambda-expression may refer to variables with automatic storage duration and the *this object (if any) of enclosing block scopes by capturing those entities, as described below.

If a lambda-capture includes a capture-default that is &,

1

no identifier in a simple-capture of that lambda-capture shall be preceded by &. If a lambda-capture includes a capture-default that is =,

2

each simple-capture of that lambda-capture shall be of the form "&identifier ...opt", "this", or "*this". [Note: 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:

```cpp
struct S2 { void f(int i); }; 
void S2::f(int i) {
  [&]{}; // OK
  [&, i]{}; // OK, equivalent to [&, 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]

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).

The identifier in a simple-capture is looked up using the usual rules for unqualified name lookup (6.5.1); 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.

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:

```cpp
void f() {
  int x = 0;
  auto g = [x](int x) { return 0; }; // error: parameter and simple-capture have the same name
}
```

—end example]

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: 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:

```cpp
int x = 4;
auto y = [&x = x, x = x+1]()->int {
  x += 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]

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 (7.6.2.1) potentially
references *this. [Note: 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 (7.6.1.7) 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.2. [Example:

```cpp
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. — end example] [Note: 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 (8.5.1). [Example:

```cpp
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);
}
```

— end example] — end note]

8 An entity is captured if it is captured explicitly or implicitly. An entity captured by a lambda-expression is odr-used (6.3) in the scope containing the lambda-expression. [Note: As a consequence, if a lambda-expression explicitly captures an entity that is not odr-usable, the program is ill-formed (6.3). — end note] [Example:

```cpp
void f1(int i) {
    int const N = 20;
    auto m1 = [=] {
        int const M = 30;
        auto m2 = [i] {
            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] {
        // error: j not odr-usable due to intervening lambda m3
        int x = n;  // error: n is odr-used but not odr-usable due to intervening lambda m3
        x += m;    // OK: m implicitly captured by m4 and explicitly captured by m3
        x += i;    // error: i is odr-used but not odr-usable
                     // due to intervening function and class scopes
        x += f;    // OK: this captured implicitly by m4 and explicitly by m3
    }
};

struct s2 {
    double ohseven = .007;
    auto f() {
        return [this] {
            return [*this] {
                return ohseven;  // OK
            }
        }();
    }
    auto g() {
        return [] {
            return [*this] {
            }
        }();
    }
};
—end example]

Note: Because local entities are not odr-usable within a default argument (6.3), a lambda-expression appearing in a default argument cannot implicitly or explicitly capture any local entity. Such a lambda-expression can still have an init-capture if any full-expression in its initializer satisfies the constraints of an expression appearing in a default argument (9.3.3.6). —end note] [Example:

void f2() {
    int i = 1;
    void g1(int = ([i] { return i; })());     // error
    void g2(int = ([i] { return 0; })());     // error
    void g3(int = ([i] { return i; })());     // error
    void g4(int = ([i] { return 0; })());     // OK
    void g5(int = ([{ return sizeof i; }])); // OK
    void g6(int = ([x=i] { return x; })());   // OK
    void g7(int = ([x=i] { return x; })());   // error
}
—end example]

An entity is captured by copy if

(10.1) — it is implicitly captured, the capture-default is =, and the captured entity is not *this, or

(10.2) — it is explicitly captured with a capture that is not of the form this, & identifier, or & identifier initializer.

For each entity captured by copy, an unnamed non-static data member is declared in the closure type. The declaration order of these members is unspecified. The type of such a data member is the referenced type if the entity is a reference to an object, an lvalue reference to the referenced function type if the entity is a reference to a function, or the type of the corresponding captured entity otherwise. A member of an anonymous union shall not be captured by copy.

Every id-expression within the compound-statement of a lambda-expression that is an odr-use (6.3) of an entity captured by copy is transformed into an access to the corresponding unnamed data member of the closure type. [Note: An id-expression that is not an odr-use refers to the original entity, never to a member of the closure type. However, such an id-expression can still cause the implicit capture of the entity. —end note] If *this is captured by copy, each expression that odr-uses *this is transformed to instead refer to the corresponding unnamed data member of the closure type. [Example:
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:

    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: 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:

    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 \( m_2 \) captures an entity and that entity is captured by an immediately enclosing lambda-expression \( m_1 \), then \( m_2 \)'s capture is transformed as follows:

(14.1) if \( m_1 \) captures the entity by copy, \( m_2 \) captures the corresponding non-static data member of \( m_1 \)'s closure type;

(14.2) if \( m_1 \) captures the entity by reference, \( m_2 \) captures the same entity captured by \( m_1 \).

[Example: The nested lambda-expressions and invocations below will output 123234.

    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: This ensures that the destructions will occur in the reverse order of the constructions. — end note]
16 [Note: 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]

17 A simple-capture containing an ellipsis is a pack expansion (13.7.3). An init-capture containing an ellipsis is a pack expansion that introduces an init-capture pack (13.7.3) whose declarative region is the lambda-expression’s compound-statement. [Example:

```c++
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 [expr.prim.fold]

1 A fold expression performs a fold of a pack (13.7.3) over a binary operator.

   fold-expression:
   ( cast-expression fold-operator ... )
   ( ... fold-operator cast-expression )
   ( cast-expression fold-operator ... fold-operator cast-expression )

   fold-operator: one of
   + - * / % ^ & | << >>
   += -= *= /= %= ^= &= |= <<= >>= =
   == != < > <= >= && || , .* ->*

2 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 (13.7.3).

3 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:

```c++
template<typename ...Args>
bool f(Args ...args) {
    return (true && ... && args); // OK
}

template<typename ...Argse>
bool f(Args ...args) {
    return (args + ... + args); // error: both operands contain unexpanded packs
}
— end example]

7.5.7 Requires expressions [expr.prim.req]

1 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_opt 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: 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::ConvertibleTo<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 (9.3.3.5). 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:

```cpp
template<typename T>
concept C = requires(T t, ...) {
  // error: terminates with an ellipsis
  t;
};
—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: 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:

```cpp
template<typename T> concept C =
  requires {
    new int[-(int)sizeof(T)]; // ill-formed, no diagnostic required
  };
—end example]

7.5.7.1 Simple requirements

simple-requirement:
  expression ;
A \textit{simple-requirement} asserts the validity of an \textit{expression}. [\textit{Note}: The enclosing \textit{requires-expression} will evaluate to \textit{false} if substitution of template arguments into the \textit{expression} fails. The \textit{expression} is an unevaluated operand (7.2). — end note] [Example:
\begin{verbatim}
template<typename T> concept C =
    requires (T a, T b) {
        a + b; // C<T> is true if a + b is a valid expression
    };
\end{verbatim}
— end example]

A \textit{requirement} that starts with a \textit{requires} token is never interpreted as a \textit{simple-requirement}. [\textit{Note}: This simplifies distinguishing between a \textit{simple-requirement} and a \textit{nested-requirement}. — end note]

\subsection*{7.5.7.2 Type requirements} [expr.prim.req.type]

\textit{type-requirement}:
\begin{verbatim}
type-name ;
\end{verbatim}

A \textit{type-requirement} asserts the validity of a \textit{type}. [\textit{Note}: The enclosing \textit{requires-expression} will evaluate to \textit{false} if substitution of template arguments fails. — end note] [Example:
\begin{verbatim}
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{verbatim}
— end example]

A \textit{type-requirement} that names a class template specialization does not require that type to be complete (6.8).

\subsection*{7.5.7.3 Compound requirements} [expr.prim.req.compound]

\textit{compound-requirement}:
\begin{verbatim}
{ expression } noexcept opt return-type-requirement opt ;
\end{verbatim}

A \textit{compound-requirement} asserts properties of the \textit{expression} \( E \). Substitution of template arguments (if any) and verification of semantic properties proceed in the following order:

\begin{enumerate}
\item[(1.1)] Substitution of template arguments (if any) into the \textit{expression} is performed.
\item[(1.2)] If the \textit{noexcept} specifier is present, \( E \) shall not be a potentially-throwing expression (14.5).
\item[(1.3)] If the \textit{return-type-requirement} is present, then:
\begin{enumerate}
\item[(1.3.1)] Substitution of template arguments (if any) into the \textit{return-type-requirement} is performed.
\item[(1.3.2)] The immediately-declared constraint (13.2) of the \textit{type-constraint} for \texttt{decltype((E))} shall be satisfied. [\textit{Example}: Given concepts \( C \) and \( D \),
\begin{verbatim}
requires {
    { E1 } -> C;
    { E2 } -> D<A_1, \cdots, A_n>;
};
\end{verbatim}
is equivalent to
\begin{verbatim}
requires {
    E1; requires C<\texttt{decltype((E1))}>>;
    E2; requires D<\texttt{decltype((E2))}, A_1, \cdots, A_n>>;
};
\end{verbatim}
(including in the case where \( n \) is zero). — end example]
\end{enumerate}
\end{enumerate}

[\textit{Example}:
\begin{verbatim}
template<typename T> concept C1 = requires(T x) {
    {x++};
};
\end{verbatim}
The compound-requirement in C1 requires that \( x++ \) is a valid expression. It is equivalent to the simple-requirement \( x++ \);

```cpp
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 \texttt{typename T::inner} is a valid type, and that \texttt{std::same_as<decltype(*x)), typename T::inner>} is satisfied.

```cpp
template<typename T> concept C3 =
    requires(T x) {
        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

7.5.7.4 Nested requirements

`expr.prim.req.nested`

A nested-requirement can be used to specify additional constraints in terms of local parameters. The constraint-expression shall be satisfied (13.5.2) 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 13.5.1. [Example:

```cpp
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 \( \text{sizeof(decltype (+t))} == 1 \) (13.5.1.2). —end example]

A local parameter shall only appear as an unevaluated operand (7.2) within the constraint-expression. [Example:

```cpp
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

`expr.compound`

7.6.1 Postfix expressions

`expr.post`

Postfix expressions group left-to-right.

```cpp
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
type-name-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 )
```
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 E1[E2] is identical (by definition) to *(E1)+(E2)), 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 E1 is sequenced before the expression E2.

[Note: A comma expression (7.6.20) appearing as the expr-or-braced-init-list of a subscripting expression is deprecated; see D.3. — end note]

3 [Note: 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 9.3.3.4 for details of array types. — end note]

4 A braced-init-list shall not be used with the built-in subscript operator.

#### 7.6.1.2 Function call

A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of initializers which constitute the arguments to the function. [Note: 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 (7.3.3) is suppressed on the postfix expression), or have function pointer type.

For a call to a non-static member function, the postfix expression shall be an implicit (11.4.2, 11.4.8) or explicit class member access (7.6.1.4) 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: A member function call of the form f() is interpreted as (*this).f() (see 11.4.2). — 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 (11.7.2) in the dynamic type of the object expression is called; such a call is referred to as a virtual function call. [Note: The dynamic type is the type of the object referred to by the current value of the object expression. 11.10.4 describes the behavior of virtual function calls when the object expression refers to an object under construction or destruction. — end note]

4 [Note: 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 (7.5.4.3), 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 (7.6.1.4, 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 (9.3.3.5) is initialized (9.4, 11.4.4.2) with its corresponding argument. If there is no corresponding argument, the default argument for the parameter is used. [Example: 

```c
template<typename ...T> int f(int n = 0, T ...t);
int x = f<int>(); // error: no argument for second function parameter
```

58) This is true even if the subscript operator is used in the following common idiom: `x[k].`
When there is no parameter for a given argument, the argument is passed in such a way that the receiving
A function can be declared to accept fewer arguments (by declaring default arguments (9.3.3.6)) or more
The result of a function call is the result of the possibly-converted operand of the
After performing the initializations, the value of an argument, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter. [Note: All side effects of argument evaluations are sequenced before the function is entered (see 6.9.1). — end note] [Example:

```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: If an operator function is invoked using operator notation, argument evaluation is sequenced as specified for the built-in operator; see 12.4.1.2. — end note] [Example:

```cpp
struct S {
    S(int);
};
int operator<<(S, int);
int i, j;
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 (8.7.3) 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: 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 (9.3.3.2); 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 (9.2.8, 5.13, 5.13.5, 9.3.3.4, 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 (9.3.3.6)) or more arguments (by using the ellipsis, ..., or a function parameter pack (9.3.3.5)) than the number of parameters in the function definition (9.5). [Note: 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: This paragraph does not apply to arguments passed to a function parameter pack. Function parameter packs are expanded during template instantiation (13.7.3), thus each such argument has a corresponding parameter when a function template specialization is actually called. — end note] The value-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are performed on the argument expression. An argument that has type cv std::nullptr_t is converted to type void* (7.3.11). 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 (11.4.3), 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 (7.3.6), or a floating-point type that is subject to the floating-point promotion (7.3.7), 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.

7.6.1.3 Explicit type conversion (functional notation) [expr.type.conv]

A simple-type-specifier (9.2.8.2) 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 (12.4.1.8) for the remainder of this subclause.

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.

7.6.1.4 Class member access [expr.ref]

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 remainder of 7.6.1.4 will address only the first option (dot). 59 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.

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 7.6.1.4 will address only the first option (dot).

Abbreviating postfix-expression.id-expression as E1.E2, E1 is called the 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 “function of () returning void”.

— Note: This value can only be used for a notional function call (7.5.4.3).

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. [Note: 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 id-expression shall name a member of the class or of one of its base classes. [Note: 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] [Note: 6.5.5 describes how names are looked up after the . and -> operators. — end note]

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 7.6.1.4, cq represents either const or the absence of const and vq represents either volatile or the absence of volatile. cv represents an arbitrary set of cv-qualifiers, as defined in 6.8.3.

If E2 is declared to have type “reference to T”, then E1.E2 is an lvalue; the type of E1.E2 is T. Otherwise, one of the following rules applies.

1. 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.
2. If E2 is a non-static data member and the type of E1 is “cq1 vq1 X”, and the type of E2 is “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 xvalue. Let the notation vq12 stand for the “union” of vq1 and vq2; that is, if vq1 or vq2 is volatile, then vq12 is volatile.

59) 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.

60) Note that (∗(E1)) is an lvalue.
Similarly, let the notation $cq_{12}$ stand for the “union” of $cq_1$ and $cq_2$; that is, if $cq_1$ or $cq_2$ is const, then $cq_{12}$ is const. If $E2$ is declared to be a mutable member, then the type of $E1.E2$ is “$cq_{12}$ $T$”. If $E2$ is not declared to be a mutable member, then the type of $E1.E2$ is “$cq_{12}$ $vq_{12}$ $T$”.

(6.3) — 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 $E2$.

(6.3.1) — If $E2$ refers to a static member function, $E1.E2$ is an lvalue.

(6.3.2) — 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 (11.4.1). [Note: Any redundant set of parentheses surrounding the expression is ignored (7.5.3). — end note]

(6.4) — If $E2$ is a nested type, the expression $E1.E2$ is ill-formed.

(6.5) — 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 $T$.

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 (11.9.2) of $E2$. [Note: The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 11.9.2. — end note]

7.6.1.5 Increment and decrement

[expr.post.incr]

1 The value of a postfix ++ expression is the value of its operand. [Note: 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 $cv$ $bool$, or a pointer to a complete object type. An operand with volatile-qualified type is deprecated; see D.5. 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: 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.

2 The operand of postfix -- is decremented analogously to the postfix ++ operator. [Note: For prefix increment and decrement, see 7.6.2.2. — end note]

7.6.1.6 Dynamic cast

[expr.dynamic.cast]

1 The result of the expression $\text{dynamic} \_\text{cast}<T>(v)$ 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 $cv$ void”. The $\text{dynamic} \_\text{cast}$ operator shall not cast away constness (7.6.1.10).

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 $cv1$ $B$” and $v$ has type “pointer to $cv2$ $D$” 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 $cv1$ $B$” and $v$ has type $cv2$ $D$ such that $B$ is a base class of $D$, the result is the unique $B$ subobject of the $D$ object referred to by $v$.61 In both the pointer and reference cases, the program is ill-formed if $B$ is an inaccessible or ambiguous base class of $D$. [Example:

```c
struct $B$ { }
struct $D$ : $B$ { }
void foo($D* dp$) {
  $B* bp = \text{dynamic} \_\text{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 (11.7.2).

61) 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.
If v is a null pointer value, the result is a null pointer value.

If T is "pointer to 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.

If C is the class type to which T points or refers, the runtime check logically executes as follows:

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.
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.
3. Otherwise, the runtime check fails.

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 (17.7.3).

Example:

```c++
class A { virtual void f(); }
class B { virtual void g(); }
class D : public virtual A, private B { }
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); // fails
    ap = dynamic_cast<A*>(&d); // succeeds
    bp = dynamic_cast<B*>(&d); // ill-formed (not a runtime check)
}

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* epi = dynamic_cast<E*>(ap); // succeeds
}
```

—end example] [Note: Subclause 11.10.4 describes the behavior of a dynamic_cast applied to an object under construction or destruction. — end note]

7.6.1.7 Type identification [expr.typeid]

The result of a typeid expression is an lvalue of static type const std::type_info (17.7.2) 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 17.7.2. 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 (11.7.2), 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 (6.8.2), the typeid expression throws an exception (14.2) of a type that would match a handler of type std::bad_typeid exception (17.7.4).

62) The recommended name for such a class is extended_type_info.
63) If p is an expression of pointer type, then *p, (**p), (**p), (**(p)), and so on all meet this requirement.
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. Value-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) conversions are not applied to the expression. If the expression is a prvalue, the temporary materialization conversion (7.3.4) 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: The `type-id` cannot denote a function type with a `cv-qualifier-seq` or a `ref-qualifier` (9.3.3.5). —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:

```c++
    class D { /* ... */ };
    D d1;
    const D d2;

    typeid(d1) == typeid(d2); // yields true
    typeid(D) == typeid(const D); // yields true
    typeid(D) == typeid(d2); // yields true
    typeid(D) == typeid(const D&); // yields true

    —end example]
```

If the header `<typeinfo>` (17.7.2) is not imported or included prior to a use of `typeid`, the program is ill-formed.

[Note: Subclause 11.10.4 describes the behavior of `typeid` applied to an object under construction or destruction. —end note]

### 7.6.1.8 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 (7.6.1.10).

An lvalue of type "cvl 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, `cvl`. 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 (7.3.11), the program is ill-formed. An xvalue of type "cvl B" can be cast to type “rvalue reference to cv2 D" with the same constraints as for an lvalue of type “cvl B". If the object of type "cvl 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:

```c++
    struct B { }
    struct D : public B { };
    D d;
    B &br = d;

    static_cast<D&>(br); // produces lvalue denoting the original d object

    —end example]
```

An lvalue of type “cvl T1” can be cast to type “rvalue reference to cv2 T2" if “cv2 T2" is reference-compatible with “cvl T1" (9.4.3). 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 (7.3.1) 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 (12.4.3.1) 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 (12.4.2), or if `T` is an aggregate type (9.4.1) 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

```c++
    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: The conversion is ill-formed when attempting to convert an expression of class type to an inaccessible or ambiguous base class. —end note] [Note: If T is “array of unknown bound of U”, this direct-initialization defines the type of the expression as U[1]. —end note]

5 Otherwise, the static_cast shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a static_cast.

6 Any expression can be explicitly converted to type cv void, in which case it becomes a discarded-value expression (7.2). [Note: 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]

7 The inverse of any standard conversion sequence (7.3) not containing an lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) 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:

```cpp
struct B { };
struct D : private B { };

void f() {
  static_cast<B*>(0);
  static_cast<int B::*>(0);
  static_cast<D*>(0);
  static_cast<int D::*>(0);
}

—end example]

8 The lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) conversions are applied to the operand. Such a static_cast is subject to the restriction that the explicit conversion does not cast away constness (7.6.1.10), and the following additional rules for specific cases:

9 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.

10 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 (7.3.10), and subsequently to the enumeration type.

11 A prvalue of type “pointer to cv1 B”, where B is a class type, can be converted to a prvalue of type “pointer to cv2 D”, where D is a complete 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 (7.3.11), the program is ill-formed. The null pointer value (6.8.2) is converted to the null pointer value of the destination type. If the prvalue of type “pointer to cv1 B” points to a B that is actually a subobject of an object of type D, the resulting pointer points to the enclosing object of type D. Otherwise, the behavior is undefined.

12 A prvalue of type “pointer to member of B of type cv1 T” can be converted to a prvalue of type “pointer to member of B of type cv2 T”, where D is a complete class type and B is a base class (11.7) of D, if cv2 is the same cv-qualification as, or greater cv-qualification than, cv1. [Note: Function types (including those used in pointer-to-member-function types) are never cv-qualified (9.3.3.5). —end note] If no valid standard conversion from “pointer to member of B of type T” to “pointer to member of D of type T” exists (7.3.12), the program is ill-formed. The null member pointer value (7.3.12) is converted to the null member pointer value of the destination type. If class 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. [Note: Although class B need not contain the original member, the dynamic type of the object with which indirection through the pointer to member is performed must contain the original member; see 7.6.4. —end note]
A prvalue of type “pointer to cv1 void” can be converted to a prvalue of type “pointer to cv2 T”, where T is an object type and cv2 is the same cv-qualification as, or greater cv-qualification than, cv1. If the original pointer value represents the address A of a byte in memory and A does not satisfy the alignment requirement of T, then the resulting pointer value is unspecified. Otherwise, if the original pointer value points to an object a, and there is an object b of type T (ignoring cv-qualification) that is pointer-interconvertible (6.8.2) with a, the result is a pointer to b. Otherwise, the pointer value is unchanged by the conversion. [Example:

```cpp
T* p1 = new T;
const T* p2 = static_cast<const T*>(static_cast<void*>(p1));
bool b = p1 == p2;  // b will have the value true.
```
— end example]

### 7.6.1.9 Reinterpret cast

The result of the expression `reinterpret_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 and the value-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) 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`.

The `reinterpret_cast` operator shall not cast away constness (7.6.1.10). 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: The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value. — end note]

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: 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: A `reinterpret_cast` cannot be used to convert a value of any type to the type `std::nullptr_t`. — end note]

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: Except as described in 6.7.5.4.3, the result of such a conversion will not be a safely-derived pointer value. — end note]

A function pointer can be explicitly converted to a function pointer of a different type. [Note: The effect of calling a function through a pointer to a function type (9.3.3.5) that is not the same as the type used in the definition of the function is undefined (7.6.1.2). — 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: See also 7.3.11 for more details of pointer conversions. — end note]

An object pointer can be explicitly converted to an object pointer of a different type. When a prvalue v of object pointer type is converted to the object pointer type “pointer to cv T”, the result is `static_cast<cv T*>(static_cast<cv void*>(v))`. [Note: Converting a prvalue of type “pointer to T1” to the type “pointer to T2” (where T1 and T2 are object types and where 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.

The null pointer value (6.8.2) is converted to the null pointer value of the destination type. [Note: 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]

---

64) The types may have different cv-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.

§ 7.6.1.9 118
A prvalue of type “pointer to member of \( X \) of type \( T_1 \)” can be explicitly converted to a prvalue of a different type “pointer to member of \( Y \) of type \( T_2 \)” if \( T_1 \) and \( T_2 \) are both function types or both object types.\(^{65}\) The null member pointer value (7.3.12) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

\(^{10}\)

- 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.

\(^{10.1}\)

- 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.

\(^{11}\)

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 reinterpret_cast. The result is that of \(*\text{reinterpret\_cast<T2>}(p)\) where \( p \) is a pointer to \( x \) of type “pointer to \( T_1 \)”. No temporary is created, no copy is made, and no constructors (11.4.4) or conversion functions (11.4.7) are called.\(^{66}\)

### 7.6.1.10 Const cast

The result of the expression \( \text{const\_cast<T>(v)} \) is of type \( T \). If \( T \) is an lvalue reference to object type, the result is an lvalue; if \( T \) is an lvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-prvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are performed on the expression \( v \). Conversions that can be performed explicitly using \( \text{const\_cast} \) are listed below. No other conversion shall be performed explicitly using \( \text{const\_cast} \).

\(^{2}\)

[Note: Subject to the restrictions in this subclause, an expression may be cast to its own type using a const_cast operator. — end note]

For two similar types \( T_1 \) and \( T_2 \) (7.3.5), a prvalue of type \( T_1 \) may be explicitly converted to the type \( T_2 \) using a const_cast if, considering the cv-decompositions of both types, each \( P_1^n \) is the same as \( P_2^n \) for all \( i \). The result of a const_cast refers to the original entity. [Example:

\[
\begin{align*}
\text{typedef int *A[3];} & \quad \text{// array of 3 pointer to int} \\
\text{typedef const int *const CA[3];} & \quad \text{// array of 3 const pointer to int} \\
\text{CA &&r = A{}}; & \quad \text{// OK, reference binds to temporary array object} \\
\text{A &&r1 = const\_cast<\&>(CA{});} & \quad \text{// after qualification conversion to type CA} \\
\text{A &&r2 = const\_cast<\&\&>(CA{});} & \quad \text{// OK} \\
\text{— end example]
\end{align*}
\]

\(^{3}\)

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 const_cast, then the following conversions can also be made:

\(^{4}\)

- an lvalue of type \( T_1 \) can be explicitly converted to an lvalue of type \( T_2 \) using the cast \( \text{const\_cast<T2\&>} \);

\(^{4.1}\)

- a glvalue of type \( T_1 \) can be explicitly converted to an xvalue of type \( T_2 \) using the cast \( \text{const\_cast<T2\&\&>} \); and

\(^{4.2}\)

- 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 \( \text{const\_cast<T2\&\&>} \).

The result of a reference \( \text{const\_cast} \) refers to the original object if the operand is a glvalue and to the result of applying the temporary materialization conversion (7.3.4) otherwise.

\(^{5}\)

A null pointer value (6.8.2) is converted to the null pointer value of the destination type. The null member pointer value (7.3.12) is converted to the null member pointer value of the destination type.\(^{6}\)

[Note: Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a const_cast that casts away a const-qualifier\(^{67}\) may produce undefined behavior (9.2.8.1). — end note]

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 (7.3.5) of \( T_1 \) yielding \( n \) such that \( T_2 \) has a cv-decomposition of the form

\[^{65}\] \( T_1 \) and \( T_2 \) may have different cv-qualities, subject to the overall restriction that a reinterpret_cast cannot cast away constness.

\[^{66}\] This is sometimes referred to as a type pun when the result refers to the same object as the source glvalue.

\[^{67}\] const_cast is not limited to conversions that cast away a const-qualifier.
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.

[Note: Some conversions which involve only changes in cv-qualification cannot be done using \texttt{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

Expressions with unary operators group right-to-left.

\[
\text{unary-expression:} \quad \text{postfix-expression} \quad \text{unary-operator cast-expression} \quad ++ \text{ cast-expression} \quad -- \text{ cast-expression} \quad \text{sizeof unary-expression} \quad \text{sizeof } (\text{type-id}) \quad \text{sizeof } \ldots (\text{identifier}) \quad \text{offsetof (type-id)} \quad \text{nothrow-expression} \quad \text{new-expression} \quad \text{delete-expression}
\]

\[
\text{unary-operator: one of} \quad \ast \quad & \quad + \quad - \quad ! \quad ~
\]

### 7.6.2.1 Unary operators

The unary \( \ast \) operator performs \textit{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: Indirection through a pointer to an incomplete type (other than \texttt{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 7.3.1. —end note]

The result of each of the following unary operators is a prvalue.

The result of the unary \& operator is a pointer to its operand.

1. If the operand is a \textit{qualified-id} naming a non-static or variant member \( m \) of some class \( C \) with type \( T \), the result has type “pointer to member of class \( C \) of type \( T \)” and is a prvalue designating \( C :: m \).

2. Otherwise, if the operand is an lvalue of type \( T \), the resulting expression is a prvalue of type “pointer to \( T \)” whose result is a pointer to the designated object (6.7.1) or function. [Note: In particular, taking the address of a variable of type “\texttt{cv T}” yields a pointer of type “pointer to \texttt{cv T}”. —end note]

3. Otherwise, the program is ill-formed.

[Example:

```
struct A { int i; };  
struct B : A { };  
... &B::i ...  // has type int A::*
int a;  
int* p1 = &a;  
int* p2 = p1 + 1;  // defined behavior  
bool b = p2 > p1;  // defined behavior, with value true
```

—end example] [Note: A pointer to member formed from a \texttt{mutable} non-static data member (9.2.1) does not reflect the \texttt{mutable} specifier associated with the non-static data member. —end note]
A pointer to member is only formed when an explicit & is used and its operand is a qualified-id not enclosed in parentheses. [Note: 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 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” (7.3.3). Nor is &unqualified-id a pointer to member, even within the scope of the unqualified-id’s class. — end note]

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: 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]

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.

The operand of the unary - operator shall have arithmetic or unscoped enumeration type and the result is the negation 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.

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.

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: 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]

7.6.2.2 Increment and decrement [expr.pre.incr]

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.5. 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: See the discussions of addition (7.6.6) and assignment operators (7.6.19) for information on conversions. — end note]

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: For postfix increment and decrement, see 7.6.1.5. — end note]

7.6.2.3 Await [expr.await]

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.

```cpp
await-expression:
  co_await cast-expression
```

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 a for-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 (9.3.3.6). An await-expression shall not appear in the initializer of a block-scope 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.

Evaluation of an await-expression involves the following auxiliary types, expressions, and objects:

- \(p\) is an lvalue naming the promise object (9.5.4) of the enclosing coroutine and \(P\) is the type of that object.
\((3.2)\)  
a is the cast-expression if the await-expression was implicitly produced by a yield-expression (7.6.17), an initial suspend point, or a final suspend point (9.5.4). Otherwise, the unqualified-id await_transform is looked up within the scope of P by class member access lookup (6.5.5), and if this lookup finds at least one declaration, then a is p.await_transform(cast-expression); otherwise, a is the cast-expression.

\((3.3)\)  
o is determined by enumerating the applicable operator co_await functions for an argument a (12.4.1.2), and choosing the best one through overload resolution (12.4). If overload resolution is ambiguous, the program is ill-formed. If no viable functions are found, o is a. Otherwise, o is a call to the selected function with the argument a. If o would be a prvalue, the temporary materialization conversion (7.3.4) is applied.

\((3.4)\)  
e is an lvalue referring to the result of evaluating the (possibly-converted) o.

\((3.5)\)  
h is an object of type std::coroutine_handle<P> referring to the enclosing coroutine.

\((3.6)\)  
await-ready is the expression e.await_ready(), contextually converted to bool.

\((3.7)\)  
await-suspend is the expression e.await_suspend(h), which shall be a prvalue of type void, bool, or std::coroutine_handle<Z> for some type Z.

\((3.8)\)  
await-resume is the expression e.await_resume().

4 The await-expression has the same type and value category as the await-resume expression.

5 The await-expression evaluates the (possibly-converted) o expression and the await-ready expression, then:

\((5.1)\)  
If the result of await-ready is false, the coroutine is considered suspended. Then:

\((5.1.1)\)  
If the type of await-suspend is std::coroutine_handle<Z>, await-suspend.resume() is evaluated.  
[Note: This resumes the coroutine referred to by the result of await-suspend. Any number of coroutines may be successively resumed in this fashion, eventually returning control flow to the current coroutine caller or resumer (9.5.4). — end note]

\((5.1.2)\)  
Otherwise, if the type of await-suspend is bool, await-suspend is evaluated, and the coroutine is resumed if the result is false.

\((5.1.3)\)  
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).

\((5.2)\)  
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.

6 [Example:

```cpp
template<typename T>
struct my_future {
  // ... *
  bool await_ready();
  void await_suspend(std::coroutine_handle<>);
  T await_resume();
};

template<typename Rep, class Period>
auto operator co_await(std::chrono::duration<Rep, Period> d) {
  struct waiter {
    std::chrono::system_clock::duration duration;
    // ... */
    waiter(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 waiter(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

7.6.2.4 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 (6.8.1) is implementation-defined. [Note: In particular, sizeof(bool), sizeof(char16_t), sizeof(char32_t), and sizeof(wchar_t) are implementation-defined. — end note] [Note: 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.99 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 (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are not applied to the operand of sizeof. If the operand is a prvalue, the temporary materialization conversion (7.3.4) is applied.

4 The identifier in a sizeof... expression shall name a pack. The sizeof... operator yields the number of elements in the pack (13.7.3). A sizeof... expression is a pack expansion (13.7.3). [Example:

    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: 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]

7.6.2.5 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: 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.

7.6.2.6 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 )

--- end example]  

--- end example]
2 The result of the noexcept operator is a prvalue of type bool. [Note: 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).

7.6.2.7 New [expr.new]

The new-expression attempts to create an object of the type-id (9.3.1) 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, 11.7.3). [Note: Because references are not objects, references cannot be created by new-expressions. — end note] [Note: The type-id may be a cv-qualified type, in which case the object created by the new-expression has a cv-qualified type. — end note]

```
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
  noptr-new-declarator [ constant-expression ] attribute-specifier-seq_opt

new-initializer:
  ( expression-list_opt )
  braced-init-list
```

2 If a placeholder type (9.2.8.5) 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 (9.2.8.5):

```
T x init;
```

[Example:

```
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>
```

—end example]

3 The new-type-id in a new-expression is the longest possible sequence of new-declarators. [Note: This prevents ambiguities between the declarator operators &. &&, *, and [] and their expression counterparts. — end note] [Example:

```
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]

4 [Note: Parentheses in a new-type-id of a new-expression can have surprising effects. [Example:

```
new int(*[10])(); // error
```

is ill-formed because the binding is

```
(new int) (*[10])(); // error
```

Instead, the explicitly parenthesized version of the new operator can be used to create objects of compound types (6.8.2):

```
new (int (*[10])());
```

allocates an array of 10 pointers to functions (taking no argument and returning int). — end example] — end note]

§ 7.6.2.7
Objects created by a `new-expression` have dynamic storage duration (6.7.5.4). [Note: 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.

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: Both `new int` and `new int[10]` have type `int*` and the type of `new int[i][10]` is `int(*)[10]` — end note] The attribute-specifier-seq in a `noptr-new-declarator` appertains to the associated array type.

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: Given the definition `int n = 42, new float[n][5]` 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]

If the `type-id` or `new-type-id` denotes an array type of unknown bound (9.3.3.4), 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` (9.4.1, 9.4.2).

If the `expression` in a `noptr-new-declarator` is present, it is implicitly converted to `std::size_t`. The `expression` is erroneous if:

1. The `expression` is of non-class type and its value before converting to `std::size_t` is less than zero;
2. The `expression` is of class type and its value before application of the second standard conversion (12.4.3.1.2) is less than zero;
3. Its value is such that the size of the allocated object would exceed the implementation-defined limit (Annex B); or
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`) exceeds the number of elements to initialize.

If the `expression` is erroneous after converting to `std::size_t`:

5. If the `expression` is a core constant expression, the program is ill-formed;
6. Otherwise, an allocation function is not called; instead
   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;
   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` (17.6.3.2).

When the value of the `expression` is zero, the allocation function is called to allocate an array with no elements.

A `new-expression` may obtain storage for the object by calling an allocation function (6.7.5.4.1). If the `new-expression` terminates by throwing an exception, it may release storage by calling a deallocation function (6.7.5.4.2). 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: An implementation is required to provide default definitions for the global allocation functions (6.7.5.4, 17.6.2.1, 17.6.2.2). A C++ program can provide alternative definitions of these functions (16.5.4.6) and/or class-specific versions (11.12). The set of allocation and deallocation functions that may be called by a `new-expression` may include functions that do not perform allocation or deallocation; for example, see 17.6.2.3. — end note]

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 (17.6.2.1, 17.6.2.2). 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: 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]

---

70) 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.
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:

14.1. the evaluation of `e1` is sequenced before the evaluation of `e2`, and
14.2. `e2` is evaluated whenever `e1` obtains storage, and
14.3. both `e1` and `e2` invoke the same replaceable global allocation function, and
14.4. 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
14.5. the pointer values produced by `e1` and `e2` are operands to evaluated `delete-expressions`, and
14.6. the evaluation of `e2` is sequenced before the evaluation of the `delete-expression` whose operand is the pointer value produced by `e1`.

**Example:**

```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;
    }
}
```

15 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 (17.6.2.3). 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: 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]

16 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.

17 The `new-placement` syntax is used to supply additional arguments to an allocation function; such an expression is called a placement `new-expression`.

18 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;
— 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.

19

[Example:

— `new T` results in one of the following calls:
  - `operator new(sizeof(T))`
  - `operator new(sizeof(T), std::align_val_t(alignof(T)))`

— `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)`

— `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)))`

— `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

20

[Note: 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 (6.7.5.4.1, Clause 14, 17.6.3.1); 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 (17.6.2.3) 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.

21

[Note: 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]

22

A `new-expression` that creates an object of type `T` initializes that object as follows:

— If the `new-initializer` is omitted, the object is default-initialized (9.4). [Note: If no initialization is performed, the object has an indeterminate value. — end note]

— Otherwise, the `new-initializer` is interpreted according to the initialization rules of 9.4 for direct-initialization.

23

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`.

24

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 (11.4.4) selected for the initialization (if any). If the `new-expression` creates an array of objects of class type, the destructor is potentially invoked (11.4.6).

25

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: This is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a memory leak. — end note]

26

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.

27 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 (9.3.3.5), 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 (7.6.2.8). [Example:

```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]

28 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.

7.6.2.8 Delete `{expr.delete}`

1 The `delete-expression` operator destroys a most derived object (6.7.2) or array created by a `new-expression`. `delete-expression`:

```c
::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 `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.

2 If the operand has a class type, the operand is converted to a pointer type by calling the above-mentioned conversion function, and the converted operand is used in place of the original operand for the remainder of this subclause. In a single-object delete expression, the value of the operand of `delete` may be a null pointer 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`. If not, the behavior is undefined. [Note: This means that the syntax of the `delete-expression` may be a null pointer value or a pointer value that resulted from a previous array `new-expression`. — end note] [Note: A pointer to a `const` type can be the operand of a `delete-expression`; it is not necessary to cast away the constness (7.6.1.10) 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

72) 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.

73) This implies that an object cannot be deleted using a pointer of type `void*` because `void` is not an object type.

74) 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.
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.

[Note: An implementation provides default definitions of the global deallocation functions operator delete for non-arrays (17.6.2.1) and operator delete[] for arrays (17.6.2.2). A C++ program can provide alternative definitions of these functions (16.5.4.6), and/or class-specific versions (11.12). —end note]

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.

If deallocation function lookup finds more than one usual deallocation function, the function to be called is selected as follows:

1. 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.
2. 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.
3. If exactly one function remains, that function is selected and the selection process terminates.
4. If the deallocation functions have class scope, the one without a parameter of type std::size_t is selected.
5. 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.
6. 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. [Note: If the deallocation function is not a destroying operator delete and the deleted object is not the most derived object in the former case, the behavior is undefined, as stated above. —end note] 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

§ 7.6.2.8
the deleted object suitably adjusted for the array allocation overhead (7.6.2.7) in an array delete expression, as its first argument. [Note: Any cv-qualifiers in the type of the deleted object are ignored when forming this argument. — end note] 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. [Note: If this results in a call to a replaceable deallocation function or the second or third argument was not the corresponding argument in said call, the behavior is undefined (17.6.2.1, 17.6.2.2). — end note]

Access and ambiguity control are done for both the deallocation function and the destructor (11.4.6, 11.12).

7.6.3 Explicit type conversion (cast notation) [expr.cast]

1 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. [Note: If T is a non-class type that is cv-qualified, the cv-qualifiers are discarded when determining the type of the resulting prvalue; see 7.2. — end note]

2 An explicit type conversion can be expressed using functional notation (7.6.1.3), a type conversion operator (dynamic_cast, static_cast, reinterpret_cast, const_cast), or the cast notation.

3 Any type conversion not mentioned below and not explicitly defined by the user (11.4.7) is ill-formed.

4 The conversions performed by

(4.1) — a const_cast (7.6.1.10),
(4.2) — a static_cast (7.6.1.8),
(4.3) — a static_cast followed by a const_cast,
(4.4) — a reinterpret_cast (7.6.1.9), or
(4.5) — 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:

(4.6) — 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;
(4.7) — 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;
(4.8) — 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:

```
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
```
The 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: 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 \( \rightarrow* \) and \( .* \) group left-to-right.

\[
\text{pm-expression:} \\
\quad \text{cast-expression} \\
\quad \text{pm-expression} .* \text{cast-expression} \\
\quad \text{pm-expression} \rightarrow* \text{cast-expression}
\]

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.

The binary operator \( \rightarrow* \) 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 \( E1\rightarrow*E2 \) is converted into the equivalent form \( *(E1)).*E2 \).

Abbreviating \( \text{pm-expression} .*\text{cast-expression} \) 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 \( E2 \).

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 7.6.1.4. [Note: It is not possible to use a pointer to member that refers to a mutable member to modify a const class object. For example,

\[
\begin{align*}
\text{struct S} & \{ \\
S() : i(0) \& \{ \\
& \text{mutable int i;} \\
\}; \\
\text{void f()} \{ \\
& \text{const S cs;} \\
& \text{int S::* pm = &S::i;} \quad // \text{pm refers to mutable member S::i} \\
& \text{cs.*pm = 88;} \quad \quad // \text{error: cs is a const object} \\
\}
\end{align*}
\]

—end note]

If the result of \( .* \) or \( \rightarrow* \) is a function, then that result can be used only as the operand for the function call operator \( () \). [Example:

\[
\text{(ptr_to_obj}&&\text{ptr_to_mfct)(10);}
\]

calls the member function denoted by ptr_to_mfct for the object pointed to by ptr_to_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 (7.3.12), the behavior is undefined.

7.6.5 Multiplicative operators

The multiplicative operators \( \ast, /, \text{and} \% \) group left-to-right.
The operands of * and / 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 / 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 / or % is zero the behavior is undefined. For integral operands the / operator yields the algebraic quotient with any fractional part discarded;\(^{75}\) if the quotient \(a/b\) is representable in the type of the result, \((a/b)\times b + a\% b\) 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.

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 \(P\).

When two pointer expressions \(P\) and \(Q\) are subtracted, the type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as std::ptrdiff_t in the <cstddef> header (17.2.4).

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 \(P\).

### 7.6.6.6 Additive operators

---

\(^{75}\) This is often called truncation towards zero.

\(^{76}\) As specified in 6.8.2, 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.
For addition or subtraction, if the expressions P or Q have type “pointer to cv T”, where T and the array element type are not similar (7.3.5), the behavior is undefined. [Note: In particular, a pointer to a base class cannot be used for pointer arithmetic when the array contains objects of a derived class type. — end note]

7.6.7 Shift operators [expr.shift]

The shift operators << and >> group left-to-right.

```cpp
shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression
```

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.

The value of E1 << E2 is the unique value congruent to E1 × 2E2 modulo 2N, where N is the width of the type of the result. [Note: E1 is left-shifted E2 bit positions; vacated bits are zero-filled. — end note]

The value of E1 >> E2 is E1/2E2, rounded down. [Note: E1 is right-shifted E2 bit positions. Right-shift on signed integral types is an arithmetic right shift, which performs sign-extension. — end note]

The expression E1 is sequenced before the expression E2.

7.6.8 Three-way comparison operator [expr.spaceship]

The three-way comparison operator groups left-to-right.

```cpp
compare-expression:
  shift-expression
  compare-expression <=> shift-expression
```

The expression p <=> q is a prvalue indicating whether p is less than, equal to, greater than, or incomparable with q.

If one of the operands is of type bool and the other is not, the program is ill-formed.

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 (9.4.4) is required, other than from an integral type to a floating-point type, the program is ill-formed.

(4.2) — 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.

(4.3) — 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 pointer type and the other operand is of pointer or array type, array-to-pointer conversions (7.3.2), pointer conversions (7.3.11), and qualification conversions (7.3.5) 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: If both of the operands are arrays, array-to-pointer conversions (7.3.2) are not applied. — end note]

If the composite pointer type is an object pointer type, p <=> q is of type std::strong_ordering. If two pointer operands p and q compare equal (7.6.10), p <=> q yields std::strong_ordering::equal; 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). 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 (9.2.8.5) 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: \texttt{a<b<c} means \texttt{(a<b)<c} and \texttt{not (a<b) || (b<c)}. — end example]

\[
\begin{align*}
\text{relational-expression:} \\
& \text{compare-expression} \\
& \text{relational-expression < compare-expression} \\
& \text{relational-expression > compare-expression} \\
& \text{relational-expression <= compare-expression} \\
& \text{relational-expression >= compare-expression}
\end{align*}
\]

The lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are performed on the operands. The comparison is deprecated if both operands were of array type prior to these conversions (D.4).

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 \texttt{false} or \texttt{true}. The type of the result is \texttt{bool}.

The usual arithmetic conversions (7.4) are performed on operands of arithmetic or enumeration type. If both operands are pointers, pointer conversions (7.3.11) and qualification conversions (7.3.5) 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\(^{77}\) is defined in terms of a partial order consistent with the following rules:

(4.1) — 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.

(4.2) — 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.

(4.3) — Otherwise, neither pointer is required to compare greater than the other.

If two operands \(p\) and \(q\) compare equal (7.6.10), \(p==q\) and \(p>=q\) both yield \texttt{true} and \(p<q\) and \(p>q\) both yield \texttt{false}. Otherwise, if a pointer \(p\) compares greater than a pointer \(q\), \(p>=q\), \(p>q\), \(q<=p\), and \(q<p\) all yield \texttt{true} and \(p<q\), \(p=q\), \(q>p\), and \(q>p\) all yield \texttt{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 operands shall yield \texttt{true} if the specified relationship is true and \texttt{false} if it is false.

7.6.10 Equality operators

Equality operators:

\[
\begin{align*}
\text{equality-expression:} \\
\text{relational-expression} \\
\text{equality-expression == relational-expression} \\
\text{equality-expression != relational-expression}
\end{align*}
\]

The \(==\) (equal to) and the \(!=\) (not equal to) operators group left-to-right. The lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are performed on the operands. The comparison is deprecated if both operands were of array type prior to these conversions (D.4).

The converted operands shall have arithmetic, enumeration, pointer, or pointer-to-member type, or type \texttt{std::nullptr_t}. The operators \(==\) and \(!=\) both yield \texttt{true} or \texttt{false}, i.e., a result of type \texttt{bool}. In each case below, the operands shall have the same type after the specified conversions have been applied.

If at least one of the operands is a pointer, pointer conversions (7.3.11), function pointer conversions (7.3.13), and qualification conversions (7.3.5) are performed on both operands to bring them to their composite pointer type (7.2.2). Comparing pointers is defined as follows:

\(^{77}\) As specified in 6.8.2, 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 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,\(^{78}\) the result of the comparison is unspecified.

Otherwise, if the pointers are both null, both point to the same function, or both represent the same address (6.8.2), they compare equal.

Otherwise, the pointers compare unequal.

If at least one of the operands is a pointer to member, pointer-to-member conversions (7.3.12), function pointer conversions (7.3.13), and qualification conversions (7.3.5) are performed on both operands to bring them to their composite pointer type (7.2.2). Comparing pointers to members is defined as follows:

- If two pointers to members are both the null member pointer value, they compare equal.
- If only one of two pointers to members is the null member pointer value, they compare unequal.
- If either is a pointer to a virtual member function, the result is unspecified.
- 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.\([\text{Example:}]\)
  ```
  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  
  -- end example  
  ```
- If both refer to (possibly different) members of the same union (11.5), they compare equal.
- 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.\([\text{Example:}]\)
  ```
  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::*pdb)() = pl;  
  int (D::*pdr)() = pr;  
  
  bool x = (pdb == pdr); // false  
  bool y = (pb == pl); // true  
  -- end example  
  ```

Two operands of type \texttt{std::nullptr_t} or one operand of type \texttt{std::nullptr_t} and the other a null pointer constant compare equal.

If two operands compare equal, the result is \texttt{true} for the \texttt{==} operator and \texttt{false} for the \texttt{!=} operator. If two operands compare unequal, the result is \texttt{false} for the \texttt{==} operator and \texttt{true} for the \texttt{!=} 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 \texttt{true} if the specified relationship is true and \texttt{false} if it is false.

### 7.6.11 Bitwise AND operator

\(\text{expr.bit.and}\)

\(^{78}\) As specified in 6.8.2, an object that is not an array element is considered to belong to a single-element array for this purpose.
7.6.12 Bitwise exclusive OR operator

```
exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ and-expression
```

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 (6.8.1) of the converted operands \(x\) and \(y\), the coefficient \(r_i\) of the base-2 representation of the result \(r\) is 1 if either \(x_i\) and \(y_i\) are 1, and 0 otherwise. [Note: The result is the bitwise exclusive OR function of the operands. — end note]

7.6.13 Bitwise inclusive OR operator

```
inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression
```

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 (6.8.1) 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. [Note: The result is the bitwise inclusive OR function of the operands. — end note]

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
  conditional-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).
If either the second or the third operand has type `void`, one of the following shall hold:

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. Both the second and the third operands have type `void`; the result is of type `void` and is a prvalue.  
   
   [Note: 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 (12.4.3.1) from each of those operands to the type of the other. [Note: 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:

   - If `E2` is an lvalue, the target type is “lvalue reference to `T2`”, subject to the constraint that in the conversion the reference must bind directly (9.4.3) to a glvalue.
   - If `E2` is an xvalue, the target type is “rvalue reference to `T2`”, subject to the constraint that the reference must bind directly.
   - 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 `T1` and `T2` are the same class type (ignoring cv-qualification) and `T2` is at least as cv-qualified as `T1`, the target type is `T2`,
     - otherwise, if `T2` is a base class of `T1`, the target type is `cv1 T2`, where `cv1` denotes the cv-qualifiers of `T1`,
     - otherwise, the target type is the type that `E2` would have after applying the lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) 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: The conversion might be ill-formed even if an implicit conversion sequence could be formed. — end note]

5. 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.

6. 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 (12.4.1.2, 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.

7. Lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

   - 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.
   - 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.
7.3 — One or both of the second and third operands have pointer type; pointer conversions (7.3.11), function pointer conversions (7.3.13), and qualification conversions (7.3.5) are performed to bring them to their composite pointer type (7.2.2). The result is of the composite pointer type.

7.4 — One or both of the second and third operands have pointer-to-member type; pointer to member conversions (7.3.12), function pointer conversions (7.3.13), and qualification conversions (7.3.5) are performed to bring them to their composite pointer type (7.2.2). The result is of the composite pointer type.

7.5 — 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 (7.6.2.3). 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:

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{};
        }
    };
    struct iterator { /* ... */
        iterator begin();
        iterator end();
    };

    my_generator<pair<int,int>> g1() {
        for (int i = i; i < 10; ++i) co_yield {i,i};
    }

    my_generator<pair<int,int>> g2() {
        for (int i = i; i < 10; ++i) co_yield make_pair(i,i);
    }

    auto f(int x = co_yield 5);
    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

A `throw-expression` is of type `void`.

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”.

§ 7.6.18
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: Code that must be executed because of an exception, but cannot completely handle the exception itself, can be written like this:

```c++
try {
    // ...
} catch (...) {
    // catch all exceptions
    // respond (partially) to exception
    throw;
    // pass the exception to some other handler
}
```

—end example]

If no exception is presently being handled, evaluating a throw-expression with no operand calls std::terminate (14.6.1).

### 7.6.19 Assignment and compound assignment operators

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: 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.5) unless the (possibly parenthesized) assignment is a discarded-value expression or an unevaluated operand.

The behavior of an expression of the form \( E_1 \text{ op } E_2 \) is equivalent to \( E_1 = E_1 \text{ op } E_2 \) except that \( E_1 \) is evaluated only once. Such expressions are deprecated if \( E_1 \) has volatile-qualified type; see D.5. 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: 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 may 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 (12.6.2.1, 12.4).

[Example:
```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
```

### 7.6.20 Comma operator

The comma operator groups left-to-right.

```cpp
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.

In contexts where comma is given a special meaning, [Example: in lists of arguments to functions (7.6.1.2) and lists of initializers (9.4) — end example] the comma operator as described in this subclause can appear only in parentheses. [Example:

```cpp
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value 5. — end example]

[Note: A comma expression appearing as the `expr-or-braced-init-list` of a subscripting expression (7.6.1.1) is deprecated; see D.3. — 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 (11.10.5) is not performed, are called constant expressions. [Note: Constant expressions can be evaluated during translation. — end note]

```cpp
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: Such a class may have a non-trivial destructor. Within this evaluation, `std::is_constant_evaluated()` (20.15.10) 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 is usable in constant expressions at a point `P` if its initializing declaration `D` is reachable from `P` and

(4.1) it is constexpr,

(4.2) it 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

§ 7.7
A non-mutable subobject or reference member of any of the above.

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:

- **this** (7.5.2), except in a constexpr function (9.2.5) that is being evaluated as part of \( E \);
- an invocation of a non-constexpr function [Note: Overload resolution (12.4) is applied as usual. — end note];
- an invocation of an undefined constexpr function;
- an invocation of an instantiated constexpr function that fails to satisfy the requirements for a constexpr function;
- an invocation of a virtual function (11.7.2) for an object unless
  - the object is usable in constant expressions or
  - its lifetime began within the evaluation of \( E \);
- an expression that would exceed the implementation-defined limits (see Annex B);
- an operation that would have undefined behavior as specified in Clause 4 through Clause 15 of this document [Note: including, 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) — end note];
- an lvalue-to-rvalue conversion (7.3.1) 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 (7.3.1) 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 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 lambda-expression, a reference to this or to a variable with automatic storage duration defined outside that lambda-expression, where the reference would be an odr-use (6.3, 7.5.5); [Example:

```c
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
    };
}
```
— end example] [Note: If the odr-use occurs in an invocation of a function call operator of a closure type, it no longer refers to this or to an enclosing automatic variable due to the transformation (7.5.5.2) of the id-expression into an access of the corresponding data member. [Example:

```c
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(monad(2))(monad)() == monad(2)());
— end example] — end note]

— a conversion from type \( cv \ void^* \) to a pointer-to-object type;

§ 7.7
a reinterpret_cast (7.6.1.9);

— a modification of an object (7.6.19, 7.6.1.5, 7.6.2.2) 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 new-expression (7.6.2.7), unless the selected allocation function is a replaceable global allocation function (17.6.2.1, 17.6.2.2) and the allocated storage is deallocated within the evaluation of \( E \);

— a delete-expression (7.6.2.8), unless it deallocates a region of storage allocated within the evaluation of \( E \);

— a call to an instance of std::allocator<T>::allocate (20.10.10.1), unless the allocated storage is deallocated within the evaluation of \( E \);

— a call to an instance of std::allocator<T>::deallocate (20.10.10.1), unless it deallocates a region of storage allocated within the evaluation of \( E \);

— an await-expression (7.6.2.3);

— an 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) or a dynamic cast (7.6.1.6) or typeid (7.6.1.7) expression that would throw an exception;

— an asm-declaration (9.10); or

— an invocation of the va_arg macro (17.13.1).

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 of this document, or an invocation of the va_start macro (17.13.1), it is unspecified whether \( E \) is a core constant expression.

[Example:]

```cpp
int x;
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) {
  int x = k;
  // OK: not required to be a constant expression
  // because x is not constexpr
  return x;
}
constexpr int incr(int &n) {
  return ++n;
}
constexpr int g(int k) {
  constexpr int x = incr(k);
  // error: incr(k) is not a core constant expression
  // because lifetime of k began outside the expression incr(k)
  return x;
}
constexpr int h(int k) {
  int x = incr(k);
  // OK: incr(k) is not required to be a core constant expression
  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)

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 20.10.10.1, 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.

7 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 \( E \).

8 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: Such expressions may be
used as bit-field lengths (11.4.9), as enumerator initializers if the underlying type is not fixed (9.7.1), and as
alignments (9.12.2). — end note]

9 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:

```c++
struct A {
  constexpr A(int i) : val(i) { }
  constexpr operator int() const { return val; }
  constexpr operator long() const { return 42; }
private:
  int val;
};
constexpr A a = alignof(int);
alignas(a) int n; // error: ambiguous conversion
struct B { int n : a; }; // error: ambiguous conversion
```

— end example]

10 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 (7.3.1),

(10.3) — array-to-pointer conversions (7.3.2),

(10.4) — function-to-pointer conversions (7.3.3),

(10.5) — qualification conversions (7.3.5),

(10.6) — integral promotions (7.3.6),

(10.7) — integral conversions (7.3.8) other than narrowing conversions (9.4.4),

(10.8) — null pointer conversions (7.3.11) from \( \text{std::nullptr_t} \),

(10.9) — null member pointer conversions (7.3.12) from \( \text{std::nullptr_t} \), and

§ 7.7
§ 7.7

— function pointer conversions (7.3.13),

and where the reference binding (if any) binds directly. [Note: Such expressions may be used in new expressions (7.6.2.7), as case expressions (8.5.2), as enumerator initializers if the underlying type is fixed (9.7.1), as array bounds (9.3.3.4), 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:

— 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,

— 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,

— if the value is of pointer-to-member-function type, it does not designate an immediate function, and

— 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:

```c
constexpr int f() { return 42; }
constexpr auto g() { return f; }
constexpr 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
```

—end example]

Note: 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:

```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] —end note]

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.

An expression or conversion is manifestly constant-evaluated if it is:

— a constant-expression, or

— the condition of a constexpr if statement (8.5.1), or

— an immediate invocation, or

— the result of substitution into an atomic constraint expression to determine whether it is satisfied (13.5.1.2), or

— the initializer of a variable that is usable in constant expressions or has constant initialization. [Example:

```c
template<bool> struct X {
};
```

Nonetheless, implementations should provide consistent results, irrespective of whether the evaluation was performed during translation and/or during program execution.

Testing this condition may involve a trial evaluation of its initializer as described above.
X<std::is_constant_evaluated()> x;  // type X<true>
int y;
const int a = std::is_constant_evaluated() ? y : 1;  // dynamic initialization to 1
double z[a];

const int b = std::is_constant_evaluated() ? 2 : y;  // static initialization to 2
int c = y + (std::is_constant_evaluated() ? 2 : y);  // dynamic initialization to y+y

constexpr int f() {
    const int n = std::is_constant_evaluated() ? 13 : 17;  // n is 13
    int m = std::is_constant_evaluated() ? 13 : 17;  // m might be 13 or 17 (see below)
    char arr[n] = {};  // char[13]
    return m + sizeof(arr);
}

int p = f();  // m is 13; initialized to 26
int q = p + f();  // m is 17 for this call; initialized to 56

[Note: A manifestly constant-evaluated expression is evaluated even in an unevaluated operand. — end note]

An expression or conversion is potentially constant evaluated if it is:

(15.1) — a manifestly constant-evaluated expression,
(15.2) — a potentially-evaluated expression (6.3),
(15.3) — an immediate subexpression of a braced-init-list,81
(15.4) — an expression of the form & cast-expression that occurs within a templated entity,82 or
(15.5) — a subexpression of one of the above that is not a subexpression of a nested unevaluated operand.

A function or variable is needed for constant evaluation if it is:

(15.6) — a constexpr function that is named by an expression (6.3) that is potentially constant evaluated, or
(15.7) — a variable whose name appears as a potentially constant evaluated expression that is either a constexpr variable or is of non-volatile const-qualified integral type or of reference type.

81) Constant evaluation may be necessary to determine whether a narrowing conversion is performed (9.4.4).
82) Constant evaluation may be necessary to determine whether such an expression is value-dependent (13.8.2.3).
8 Statements

8.1 Preamble

Except as indicated, statements are executed in sequence.

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

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: 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 S1 encloses a statement S2 if

(3.1) — S2 is a substatement of S1 (Clause 9),
(3.2) — S1 is a selection-statement or iteration-statement and S2 is the init-statement of S1,
(3.3) — S1 is a try-block and S2 is its compound-statement or any of the compound-statements of its handlers, or
(3.4) — S1 encloses a statement S3 and S3 encloses S2.

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 9.2.8.5.

[Note: 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 In the *decl-specifier-seq* of a *condition*, each *decl-specifier* shall be either a *type-specifier* or *constexpr*.

### 8.2 Labeled statement

A statement can be labeled.

```markdown
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: A label may have the same name as another declaration in the same scope or a template-parameter from an enclosing scope. Unqualified name lookup (6.5.1) ignores labels. — end note]*

2 Case labels and default labels shall occur only in *switch* statements.

### 8.3 Expression statement

Expression statements have the form

```markdown
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: 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 (8.6.1). — end note]*

### 8.4 Compound statement or block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called “block”) is provided.

```markdown
compound-statement:
  { statement-seq_opt }
statement-seq:
  statement
  statement-seq statement
```

A compound statement defines a block scope (6.4). [*Note: A declaration is a *statement* (8.8). — end note]*

### 8.5 Selection statements

Selection statements choose one of several flows of control.

```markdown
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 9.3.3 for the optional *attribute-specifier-seq* in a condition. [*Note: 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:*

```c
if (x)
  int i;
```

can be equivalently rewritten as
if (x) {
    int i;
}

Thus after the if statement, i is no longer in scope. —end example]

8.5.1 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.83

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: 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 (8.5.2) 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 (8.7.5) in the same substatement. [Example:

```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() { // no definition of x required
    if constexpr (true)
        return 0;
    else if (x)
        return x;
    else
        return -x;
}

—end example]

3 An if statement of the form

if constexpr_opt (init-statement condition) statement

is equivalent to

{ init-statement
    if constexpr_opt (condition) statement
}

and an if statement of the form

if constexpr_opt (init-statement condition) statement else statement

is equivalent to

{ init-statement
    if constexpr_opt (condition) statement else statement
}

except that names declared in the init-statement are in the same declarative region as those declared in the condition.

83) In other words, the else is associated with the nearest un-elsed if.
8.5.2 The switch statement

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 (7.3.6), 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. 8.7.1. [Note: 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

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 declspecifier-seq declarator
   attribute-specifier-seq_opt declspecifier-seq ref-qualifier_opt [ identifier-list ]

   for-range-initializer:
   expr-or-braced-init-list

See 9.3.3 for the optional attribute-specifier-seq in a for-range-declaration. [Note: An init-statement ends with a semicolon. — end note]

2 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:

   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

3 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:

```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]

8.6.1 The `while` statement

1 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.

2 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

```c
label :
{
    if (condition) {
        statement
        goto label ;
    }
}
```

[Note: The variable created in the condition is destroyed and created with each iteration of the loop. [Example:

```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]

8.6.2 The `do` statement

1 The expression is contextually converted to `bool` (7.3); if that conversion is ill-formed, the program is ill-formed.

2 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.

8.6.3 The `for` statement

1 The `for` statement

```c
for (init-statement condition_{opt} ; expression_{opt} ) statement
```

is equivalent to

§ 8.6.3 150
except that names declared in the \textit{init-statement} are in the same declarative region as those declared in the \textit{condition}, and except that a \texttt{continue} in \textit{statement} (not enclosed in another iteration statement) will execute \textit{expression} before re-evaluating \textit{condition}. \textit{Note}: 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 \texttt{false}; the expression often specifies incrementing that is sequenced after each iteration. \hfill — end note\[2\]

Either or both of the \textit{condition} and the \textit{expression} can be omitted. A missing \textit{condition} makes the implied \texttt{while} clause equivalent to \texttt{while(true)}.

If the \textit{init-statement} is a declaration, the scope of the name(s) declared extends to the end of the \texttt{for} statement. \hfill [Example:]
\begin{verbatim}
int i = 42;
int a[10];
for (int i = 0; i < 10; i++)
a[i] = i;
int j = i;    // j = 42
— end example]
\end{verbatim}

\subsection*{8.6.4 The range-based \texttt{for} statement} \hfill [stmt.ranged]

\texttt{The range-based \texttt{for} statement}

\begin{verbatim}

for (init-statement\texttt{opt} for-range-declaration : for-range-initializer) statement
\end{verbatim}

is equivalent to

\begin{verbatim}
{
init-statement\texttt{opt}
auto &&range = for-range-initializer;
auto begin = begin-exp; 
auto end = end-exp;
for ( ; begin != end; ++begin ) {
  for-range-declaration = *begin;
  statement
}
}
\end{verbatim}

where
\begin{enumerate}
\item[(1.1)] if the \texttt{for-range-initializer} is an \texttt{expression}, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two \texttt{init-declarators});
\item[(1.2)] \texttt{range}, \texttt{begin}, and \texttt{end} are variables defined for exposition only; and
\item[(1.3)] \texttt{begin-exp} and \texttt{end-exp} are determined as follows:
\begin{enumerate}
\item[(1.3.1)] if the \texttt{for-range-initializer} is an \texttt{expression} of array type \texttt{R}, \texttt{begin-exp} and \texttt{end-exp} are \texttt{range} and \texttt{range + N}, respectively, where \texttt{N} is the array bound. If \texttt{R} is an array of unknown bound or an array of incomplete type, the program is ill-formed;
\item[(1.3.2)] if the \texttt{for-range-initializer} is an \texttt{expression} of class type \texttt{C}, the \texttt{unqualified-ids} \texttt{begin} and \texttt{end} are looked up in the scope of \texttt{C} as if by class member access lookup \((6.5.5)\), and if both find at least one declaration, \texttt{begin-exp} and \texttt{end-exp} are \texttt{range.begin()} and \texttt{range.end()}, respectively;
\item[(1.3.3)] otherwise, \texttt{begin-exp} and \texttt{end-exp} are \texttt{begin(range)} and \texttt{end(range)}, respectively, where \texttt{begin} and \texttt{end} are looked up in the associated namespaces \((6.5.2)\). \textit{Note}: Ordinary unqualified lookup \((6.5.1)\) is not performed. \hfill — end note\end{enumerate}
\end{enumerate}

\section*{8.6.4}
Example:
```c
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
  x *= 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

Jump statements unconditionally transfer control.

```
jump-statement:
  break ;
  continue ;
  return expr-or-braced-init-listopt ;
  coroutine-return-statement
  goto identifier ;
```

On exit from a scope (however accomplished), objects with automatic storage duration (6.7.5.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [Note: 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: 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: A suspension of a coroutine (7.6.2.3) is not considered to be an exit from a scope. — end note]

8.7.1 The break statement

```
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.
```

8.7.2 The continue statement

```
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
```
```
while (foo) {
  do {
    {
      // ...
    } // ...
    {
      continue ;
    } // ...
  } while (foo);
}
```
```
a continue not contained in an enclosed iteration statement is equivalent to goto continue.
```

8.7.3 The return statement

```
The return statement shall occur only in a function 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 (11.4.4), or a destructor (11.4.6). 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: 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 may be elided or converted to a move operation if an automatic storage duration variable is returned (11.10.5). — end note] [Example:
```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}```
The destructor for the returned object is potentially invoked (11.4.6, 14.3). [Example:

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.

The copy-initialization of the result of the call is sequenced before the destruction of temporaries at the end of the full-expression established by the operand of the return statement, which, in turn, is sequenced before the destruction of local variables (8.7) of the block enclosing the return statement.

8.7.4 The co_return statement  

co_return-statement:
  co_return expr-or-braced-init-list_opt ;

A coroutine returns to its caller or resumer (9.5.4) by the co_return statement or when suspended (7.6.2.3). A coroutine shall not enclose a return statement (8.7.3). [Note: For this determination, it is irrelevant whether the return statement is enclosed by a discarded statement (8.5.1). — 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.

The goto statement  

go_to-statement:
  goto label_opt ;

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  

declaration-statement:
  block-declaration

A declaration statement introduces one or more new identifiers into a block; it has the form

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 (6.7.5.3) 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:

```
void f() {
  // ...
  goto lx;  // error: jump into scope of a
  // ...
  ly:
  X a = 1;
```

84) The transfer from the condition of a switch statement to a case label is considered a jump in this respect.
Dynamic initialization of a block-scope variable with static storage duration (6.7.5.1) or thread storage duration (6.7.5.2) 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.\(^{85}\) If control re-enters the declaration recursively while the variable is being initialized, the behavior is undefined. [Example:

```c
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: 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 (7.6.1.3) 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.

[Note: 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: Assuming T is a simple-type-specifier (9.2.8),

```c
T(a)->m = 7; // expression-statement
T(a)++; // expression-statement
T(a,5)<<; // 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:

```c
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]
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: This can occur only when the name is declared earlier in the declaration. — end note] [Example:

```c
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))(int), c, d;

void f() {
    // disambiguation requires this to be parsed as a declaration:
    T1(a) = 3,
    T2(4),  // T2 will be declared as a variable of type T1, but this will not
    (**b)(T2(c))(int(d));  // 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

\[
\text{declaration-seq}: \\
\text{declaration} \\
\text{declaration-seq declaration}
\]

\[
\text{declaration}: \\
\text{block-declaration} \\
\text{nodeclspec-function-declaration} \\
\text{function-definition} \\
\text{template-declaration} \\
\text{deduction-guide} \\
\text{explicit-instantiation} \\
\text{explicit-specialization} \\
\text{export-declaration} \\
\text{linkage-specification} \\
\text{namespace-definition} \\
\text{empty-declaration} \\
\text{attribute-declaration} \\
\text{module-import-declaration}
\]

\[
\text{block-declaration}: \\
\text{simple-declaration} \\
\text{asm-declaration} \\
\text{namespace-alias-definition} \\
\text{using-declaration} \\
\text{using-enum-declaration} \\
\text{using-directive} \\
\text{static_assert-declaration} \\
\text{alias-declaration} \\
\text{opaque-enum-declaration}
\]

\[
\text{nodeclspec-function-declaration}: \\
\text{attribute-specifier-seq}^{\text{opt}} \text{ declarator ;}
\]

\[
\text{alias-declaration}: \\
\text{using identifier attribute-specifier-seq}^{\text{opt}} = \text{defining-type-id ;}
\]

\[
\text{simple-declaration}: \\
\text{decl-specifier-seq init-declarator-list}^{\text{opt}} ; \\
\text{attribute-specifier-seq decl-specifier-seq init-declarator-list}^{\text{opt}} ; \\
\text{attribute-specifier-seq decl-specifier-seq ref-qualifier}^{\text{opt}} [ \text{identifer-list} ] \text{ initializer ;}
\]

\[
\text{static_assert-declaration}: \\
\text{static_assert ( constant-expression ) ;} \\
\text{static_assert ( constant-expression , string-literal ) ;}
\]

\[
\text{empty-declaration}: \\
; \\
\text{attribute-declaration}: \\
\text{attribute-specifier-seq ;}
\]

[Note: \text{asm-declarations} are described in 9.10, and \text{linkage-specifications} are described in 9.11; \text{function-defintions} are described in 9.5 and \text{template-declarations} and \text{deduction-guides} are described in 13.7.1.2; \text{namespace-defintions} are described in 9.8.1, \text{using-declarations} are described in 9.9 and \text{using-directives} are described in 9.8.3. — end note]

2 A \text{simple-declaration} or \text{nodeclspec-function-declaration} of the form

\[
\text{attribute-specifier-seq}^{\text{opt}} \text{ decl-specifier-seq}^{\text{opt}} \text{ init-declarator-list}^{\text{opt}} ;
\]

is divided into three parts. Attributes are described in 9.12. \text{decl-specifiers}, the principal components of a \text{decl-specifier-seq}, are described in 9.2. \text{declarators}, the components of an \text{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`. [Note: In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the `declarator-id` for that declaration. — end note] [Example:

```c
[[noreturn]] void f [[noreturn]] () ;  // OK
```
—end example]

3 Except where otherwise specified, the meaning of an `attribute-declaration` is implementation-defined.

4 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.

5 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 `enumerator`, 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. [Example:

```c
enum { } ;  // error
typedef class { } ; // error
```
—end example]

6 In a `static_assert-declaration`, the `constant-expression` shall be a contextually converted constant expression of type `bool` (7.7). If the value of the expression when so converted is `true`, the declaration has no effect. Otherwise, the program is ill-formed, and the resulting diagnostic message (4.1) shall include the text of the `string-literal`, if one is supplied, except that characters not in the basic source character set (5.3) are not required to appear in the diagnostic message. [Example:

```c
static_assert(sizeof(int) == sizeof(void*), "wrong pointer size");
```
—end example]

7 An `empty-declaration` has no effect.

8 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` (9.2.8.5), 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.

9 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` (9.2.8) in the `decl-specifier-seq` and the recursive declarator structure of the `init-declarator` describe a type (9.3.3), which is then associated with the name being declared by the `init-declarator`.

10 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 (9.2.3). 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 (9.3.3.5) and an `object declaration` otherwise.

11 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.

12 A `nodeclspec-function-declaration` shall declare a constructor, destructor, or conversion function. [Note: A `nodeclspec-function-declaration` can only be used in a `template-declaration` (13.1), `explicit-instantiation` (13.9.2), or `explicit-specialization` (13.9.3). — end note]

### 9.2 Specifiers

The specifiers that can be used in a declaration are
The optional attribute-specifier-seq in a decl-specifier-seq appertains to the type determined by the preceding decl-specifiers (9.3.3). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

Each decl-specifier shall appear at most once in a complete decl-specifier-seq, except that long may appear twice. At most one of the constexpr, consteval, and constinit keywords shall appear in a decl-specifier-seq.

If a type-name is encountered while parsing a decl-specifier-seq, it is interpreted as part of the 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:

definition Pc;
static Pc;    // error: name missing
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,

```c
void f(const Pc);
void g(const int Pc);
```
—end example]

[Note: 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:

```c
void h(unsigned Pc);
void k(unsigned int Pc);
```
—end example] — end note]

## 9.2.1 Storage class specifiers

The storage class specifiers are

```c
storage-class-specifier:
  static
  thread_local
  extern
  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 (11.5.1)). 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: See 13.9.3 and 13.9.2 for restrictions in explicit specializations and explicit instantiations, respectively. — end note]

[Note: A variable declared without a storage-class-specifier at block scope or declared as a function parameter has automatic storage duration by default (6.7.5.3). — end note]

The thread_local specifier indicates that the named entity has thread storage duration (6.7.5.2). 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 (11.5.1). 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 (6.7.5.1), unless accompanied by the `thread_local` specifier, which declares the variable to have thread storage duration (6.7.5.2). A `static` specifier can be used in declarations of class members; 11.4.8 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: 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:

```
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();                // external linkage
inline void h();         // external linkage
inline void l();         // external linkage
void l();                // external linkage
inline void m();         // external linkage
extern void m();         // external linkage
static void n();         // internal linkage
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]

7 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:

```
struct S;
extern S a;
extern S f();
extern void g(S);

void h() {
    g(a);               // error: S is incomplete
```

§ 9.2.1
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:

```cpp
class X {
    mutable const int* p; // OK
    mutable int* const q; // error
};
```
—end example]

8 [Note: 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 (6.8.3, 9.2.8.1). — end note]

### 9.2.2 Function specifiers

A `function-specifier` can be used only in a function declaration.

```cpp
function-specifier:
    virtual
    explicit-specifier

explicit-specifier:
    explicit ( constant-expression )
    explicit
```

1 The `virtual` specifier shall be used only in the initial declaration of a non-static class member function; see 11.7.2.

2 An `explicit-specifier` shall be used only in the declaration of a constructor or conversion function within its class definition; see 11.4.7.1 and 11.4.7.2.

4 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`.

### 9.2.3 The typedef specifier

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: After

```cpp
typedef int MILES, *KLICKSP;
```
the constructions

```cpp
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]

2 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:}
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.

3 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:

```c
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

—end example]

4 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:

```c
struct S {
    typedef struct A { } A;  // OK
    typedef struct B B;     // OK
    typedef A A;            // error
};
```

—end example]

5 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:

```c
struct S;
typedef struct S S;
int main() {
    struct S* p;          // OK
}
struct S { };          // OK
```

—end example]

6 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:

```c
class complex { /* ... */ };    // error: redefinition
typedef int complex;            // error: redefinition
```

—end example]

7 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:

```c
typedef int complex;
class complex { /* ... */ };  // error: redefinition
```

—end example]

8 A simple-template-id is only a typedef-name if its template-name names an alias template or a template template-parameter. [Note: 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 (9.2.8.3), a class definition (Clause 11), a constructor declaration (11.4.4), or a destructor declaration (11.4.6), the program is ill-formed. — end note] [Example:

```c
struct S {
    S0;
    ~S0();
};

typedef struct S T;
S a = T();          // OK
struct T * p;       // error
```
9 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: 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:

```c
typedef struct { } *ps, S; // S is the class name for linkage purposes
typedef decltype([]()) C; // the closure type has no name for linkage purposes
```
—end example]

10 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:

```c
typedef struct {
    int f() {}
} X; // error: struct with typedef name for linkage has member functions
```
—end example]

9.2.4 The friend specifier

The friend specifier is used to specify access to class members; see 11.9.3.

9.2.5 The constexpr and consteval specifiers

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 (9.2.7). 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: An explicit specialization can differ from the template declaration with respect to the constexpr or consteval specifier. — end note] [Note: Function parameters cannot be declared constexpr. — end note] [Example:

```c
constexpr void square(int &x); // OK: declaration
constexpr int bufsz = 1024; // OK: definition
constexpr struct pixel {
    int x;
    int y;
    constexpr pixel(int);
} x; // error: struct with typedef name for linkage has member functions  
constexpr pixel small(2); // error: square not defined, so small(2) // not constant (7.7) so constexpr not satisfied
```
—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.
The definition of a constexpr function shall satisfy the following requirements:

1. its return type (if any) shall be a literal type;
2. each of its parameter types shall be a literal type;
3. it shall not be a coroutine (9.5.4);
4. if the function is a constructor or destructor, its class shall not have any virtual base classes;
5. its function-body shall not enclose (8.1)
   - a goto statement,
   - an identifier label (8.2),
   - a definition of a variable of non-literal type or of static or thread storage duration.

[Note: A function-body that is = delete or = default encloses none of the above. — end note]

[Example:
```
constexpr int square(int x)
{ return x * x; } // OK
constexpr long long_max()
{ return 2147483647; } // OK
constexpr int abs(int x) {
  if (x < 0)
    x = -x;
  return x; // OK
}
constexpr int first(int n) {
  static int value = n;
  // error: variable has static storage duration
  return value;
}
constexpr int uninitialized() {
  struct { int a; } s;
  // error: uninitialized read of s.a
  return 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:

1. for a non-delegating constructor, every constructor selected to initialize non-static data members and base class subobjects shall be a constexpr constructor;
2. for a delegating constructor, the target constructor shall be a constexpr constructor.

[Example:
```
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:

1. for every subobject of class type or (possibly multi-dimensional) array thereof, that class type shall have a constexpr destructor.

§ 9.2.5 163
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:
  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
    // lvalue-to-rvalue 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 (11.10.5).

[Note: Declaring a function constexpr can change whether an expression is a constant expression. This can indirectly cause calls to std::is_constant_evaluated within an invocation of the function to produce a different value. —end note]

9 The constexpr and consteval specifiers have no effect on the type of a constexpr function. [Example:
  constexpr 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; }
—end example]

10 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:
  struct pixel {
    int x, y;
  };
  constexpr pixel ur = { 1294, 1024 }; // OK
  constexpr pixel origin; // error: initializer missing
—end example]

9.2.6 The constinit specifier

[decl.constinit]

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. [Note: The `constinit` specifier ensures that the variable is initialized during static initialization (6.9.3.2). — end note]

Example:
```
const char * g() { return "dynamic initialization"; }
constexpr const char * f(bool p) { return p ? "constant initializer" : g(); }
constinit const char * c = f(true); // OK
constinit const char * d = f(false); // error
```

— end example

### 9.2.7 The `inline` specifier

The `inline` specifier shall be applied only to the declaration of a variable or function. A function declaration (9.3.3.5, 11.4.1, 11.9.3) 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. [Note: The `inline` keyword has no effect on the linkage of a function. In certain cases, an inline function cannot use names with internal linkage; see 6.6. — end note]

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: A call to an inline function or a use of an inline variable may be encountered before its definition becomes reachable in a translation unit. — end note]

[Note: 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: A constexpr function (9.2.5) is implicitly inline. In the global module, a function defined within a class definition is implicitly inline (11.4.1, 11.9.3). — end note]

### 9.2.8 Type specifiers

The type-specifiers are
```
type-specifier:
  simple-type-specifier
  elaborated-type-specifier
  typename-specifier
  cv-qualifier
type-specifier-seq:
  type-specifier attribute-specifier-seq
  type-specifier type-specifier-seq
defining-type-specifier:
  type-specifier
  class-specifier
  enum-specifier
defining-type-specifier-seq:
  defining-type-specifier attribute-specifier-seq
  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 (9.3.3). 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: 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 this subclause.

9.2.8.1 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. [Note: 6.8.3 and 9.3.3.5 describe how cv-qualifiers affect object and function types. — end note] Redundant cv-qualifications are ignored. [Note: For example, these could be introduced by typedefs. — end note]

2. [Note: Declaring a variable const can affect its linkage (9.2.1) 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: Cv-qualifiers are supported by the type system so that they cannot be subverted without casting (7.6.1.10). — end note]

4. Any attempt to modify (7.6.19, 7.6.1.5, 7.6.2.2) a const object (6.8.3) during its lifetime (6.7.3) results in undefined behavior. [Example:

```cpp
const int ci = 3; // cv-qualified (initialized as required)
    ci = 4;       // error: attempt to modify const
int i = 2;        // not cv-qualified
const int* cip;   // pointer to const int
    cip = &i;     // OK: cv-qualified access path to unqualified
    *cip = 4;     // error: attempt to modify through ptr to const
int* ip;
    ip = const_cast<int*>(cip);  // cast needed to convert const int* to int*
    *ip = 4;           // defined: *ip points to i, a non-const object
const int* ciq = new const int (3); // initialized as required
int* iq = const_cast<int*>(ciq); // cast required
    *iq = 4;        // undefined behavior: modifies a const object
```

For another example,

```cpp
struct X {
    mutable int i;
    int j;
};
```

86) There is no special provision for a decl-specifier-seq that lacks a type-specifier or that has a type-specifier that only specifies cv-qualifiers. The “implicit int” rule of C is no longer supported.
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.

[Note: 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]

### 9.2.8.2 Simple type specifiers

The simple type specifiers are

```plaintext
simple-type-specifier:
    nested-name-specifier opt type-name
    nested-name-specifier template simple-template-id
declay-type-specifier
placehole-type-specifier
nested-name-specifier opt template-name
char
cchar_t
char16_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 (9.2.8.5). A `type-specifier` of the form `typename opt nested-name-specifier opt template-name` is a placeholder for a deduced class type (9.2.8.6). 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

```plaintext
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: An injected-class-name is never interpreted as a `template-name` in contexts where class template argument deduction would be performed (13.8.1). — 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 (6.8.1). Table 14 summarizes the valid combinations of `simple-type-specifiers` and the types they specify.
### 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 9.2.8.4</td>
</tr>
<tr>
<td>placeholder-type-specifier</td>
<td>the type as defined in 9.2.8.5</td>
</tr>
<tr>
<td>template-name</td>
<td>the type as defined in 9.2.8.6</td>
</tr>
<tr>
<td>char</td>
<td>&quot;char&quot;</td>
</tr>
<tr>
<td>unsigned char</td>
<td>&quot;unsigned char&quot;</td>
</tr>
<tr>
<td>signed char</td>
<td>&quot;signed char&quot;</td>
</tr>
<tr>
<td>char8_t</td>
<td>&quot;char8_t&quot;</td>
</tr>
<tr>
<td>char16_t</td>
<td>&quot;char16_t&quot;</td>
</tr>
<tr>
<td>char32_t</td>
<td>&quot;char32_t&quot;</td>
</tr>
<tr>
<td>bool</td>
<td>&quot;bool&quot;</td>
</tr>
<tr>
<td>unsigned</td>
<td>&quot;unsigned int&quot;</td>
</tr>
<tr>
<td>unsigned int</td>
<td>&quot;unsigned int&quot;</td>
</tr>
<tr>
<td>signed</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>signed int</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>int</td>
<td>&quot;int&quot;</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>&quot;unsigned short int&quot;</td>
</tr>
<tr>
<td>unsigned short</td>
<td>&quot;unsigned short int&quot;</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>&quot;unsigned long int&quot;</td>
</tr>
<tr>
<td>unsigned long</td>
<td>&quot;unsigned long int&quot;</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>&quot;unsigned long long int&quot;</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>&quot;unsigned long long int&quot;</td>
</tr>
<tr>
<td>signed long int</td>
<td>&quot;long int&quot;</td>
</tr>
<tr>
<td>signed long</td>
<td>&quot;long int&quot;</td>
</tr>
<tr>
<td>signed long long int</td>
<td>&quot;long long int&quot;</td>
</tr>
<tr>
<td>signed long long</td>
<td>&quot;long long int&quot;</td>
</tr>
<tr>
<td>long long int</td>
<td>&quot;long long int&quot;</td>
</tr>
<tr>
<td>long long</td>
<td>&quot;long long int&quot;</td>
</tr>
<tr>
<td>long int</td>
<td>&quot;long int&quot;</td>
</tr>
<tr>
<td>long</td>
<td>&quot;long int&quot;</td>
</tr>
<tr>
<td>signed short int</td>
<td>&quot;short int&quot;</td>
</tr>
<tr>
<td>signed short</td>
<td>&quot;short int&quot;</td>
</tr>
<tr>
<td>short int</td>
<td>&quot;short int&quot;</td>
</tr>
<tr>
<td>short</td>
<td>&quot;short int&quot;</td>
</tr>
<tr>
<td>wchar_t</td>
<td>&quot;wchar_t&quot;</td>
</tr>
<tr>
<td>float</td>
<td>&quot;float&quot;</td>
</tr>
<tr>
<td>double</td>
<td>&quot;double&quot;</td>
</tr>
<tr>
<td>long double</td>
<td>&quot;long double&quot;</td>
</tr>
<tr>
<td>void</td>
<td>&quot;void&quot;</td>
</tr>
</tbody>
</table>

3 When multiple simple-type-specifiers are allowed, they can be freely intermixed with other decl-specifiers in any order. [Note: 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]

### 9.2.8.3 Elaborated type specifiers

elaborated-type-specifier:

- class-key attribute-specifier-seq<sub>opt</sub> nested-name-specifier<sub>opt</sub> identifier
class-key simple-template-id
class-key nested-name-specifier template<sub>opt</sub> simple-template-id
elaborated-enum-specifier

elaborated-enum-specifier:

- enum nested-name-specifier<sub>opt</sub> 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 (13.9.3), an explicit instantiation (13.9.2) 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 ;`

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: 6.5.4 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 (9.2.8.2). If the identifier or simple-template-id resolves to a typedef-name (9.2.3, 13.3), the elaborated-type-specifier is ill-formed. [Note: This implies that, within a class template with a template type-parameter T, the declaration `friend class T;` is ill-formed. However, the similar declaration `friend T;` is allowed (11.9.3). — end note]

3 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:

```c
enum class E { a, b };
enum E x = E::a; // OK
struct S { } s;
class S* p = &s; // OK
```

—end example]

### 9.2.8.4 Decltype specifiers

decltype-specifier:

```c
decltype ( expression )
```

1 For an expression E, the type denoted by `decltype(E)` is defined as follows:

- 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;
- 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 (9.2.8.5, 9.2.8.6);
- otherwise, if E is an unparenthesized id-expression or an unparenthesized class member access (7.6.1.4), `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;
- otherwise, if E is an xvalue, `decltype(E)` is `T&&`, where T is the type of E;
- otherwise, if E is an lvalue, `decltype(E)` is `T&`, where T is the type of E;
- otherwise, `decltype(E)` is the type of E.

The operand of the decltype specifier is an unevaluated operand (7.2).

[Example:

```c
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1 = 17; // type is const int&&
decltype(i) x2; // type is int
decltype(a->x) x3; // type is double
```
decltype((a->x)) x4 = x3; // type is const double*
—end example] [Note: The rules for determining types involving decltype(auto) are specified in 9.2.8.5.
—end note]

2 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 (7.3.4) and no result object is provided for the prvalue. The type of the prvalue may be incomplete or an abstract class type. [Note: 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: Unlike the preceding rule, parentheses have no special meaning in this context. —end note]

[Example:

```cpp
template<class T> struct A { ~A() = delete; };
template<class T> auto h() -> A<T>;
template<class T> auto i(T) -> T;
template<class T> auto f(T) -> 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) -> void; // #1
auto g() -> void {
    f(42); // OK: calls #2. (#1 is not a viable candidate: type deduction fails (13.10.2) because A<T>::~A() is implicitly used in its // decltype-specifier)
}
```

§ 9.2.8.5 170

9.2.8.5 Placeholder type specifiers [decl.spec.auto]

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: Having a generic parameter type placeholder signifies that the function is an abbreviated function template (9.3.3.5) 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 (9.3.3.5), 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 (8.5.1).
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 declarators, each of which shall be followed by a non-empty initializer. In an initializer of the form

( expression-list )

the expression-list shall be a single assignment-expression. [Example:

```cpp
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
g(auto { 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 (7.6.2.7) 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 this subclause 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 (9.2.8.5.1), and if the type that replaces the placeholder type is not the same in each deduction, the program is ill-formed.

[Example:

```cpp
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:

```cpp
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: 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:

```cpp
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: 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 (13.10.2). —end
note]  [Example:

```c++
template <class T> auto f(T t) { return t; }  // return type deduced at instantiation time
typedef decltype(f(1)) int_t;              // instantiates f<int> to deduce return type
template<class T> auto f(T* t) { return *t; } // instantiates both f's to determine return types,
                                                   // 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:

```c++
auto f();
auto f() { return 42; }     // return type is int
auto f();                  // OK
int f();                   // error: cannot be overloaded with auto f()
decltype(auto) f();        // error: auto and decltype(auto) don't match

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

template <class T> T g(T t) { return t; }       // OK, not functionally equivalent to #1
template char g(char);                         // OK, now there is a matching template
template auto g(float);                       // still matches #1

void h() { return g(42); }                     // error: ambiguous

template <typename T> struct A {
    friend T ffr(T);
};

auto ffr(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]

13 A function declared with a return type that uses a placeholder type shall not be virtual (11.7.2).
14 A function declared with a return type that uses a placeholder type shall not be a coroutine (9.5.4).
15 An explicit instantiation declaration (13.9.2) 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:

```c++
template <typename T> auto f(T t) { return t; }
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]
```

9.2.8.5.1 Placeholder type deduction  [dcl.type.auto.deduct]

1 Placeholder type deduction is the process by which a type containing a placeholder type is replaced by a
deduced type.
2 A type T containing a placeholder type, and a corresponding initializer E, are determined as follows:
— for a non-discarded \texttt{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 \texttt{return} statement. If the \texttt{return} statement has no operand, then \( E \) is \texttt{void()};

— 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 \texttt{braced-init-list} containing only a single \texttt{assignment-expression} and \( E \) is the \texttt{assignment-expression};

— for a non-type template parameter declared with a type that contains a placeholder type, \( T \) is the declared return type of the non-type template parameter and \( E \) is the corresponding template argument.

In the case of a \texttt{return} statement with no operand or with an operand of type \texttt{void}, \( T \) shall be either \texttt{\textup{type-constraint} opt decltype(auto)} or \texttt{cv \textup{type-constraint} opt auto}.

3 If the deduction is for a \texttt{return} statement and \( E \) is a \texttt{braced-init-list} (9.4.4), the program is ill-formed.

4 If the \texttt{placeholder-type-specifier} is of the form \texttt{\textup{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 \texttt{\textup{type-constraint} opt auto} either with a new invented type template parameter \( U \) or, if the initialization is copy-list-initialization, with \texttt{std::initializer_list<U>}. Deduce a value for \( U \) using the rules of template argument deduction from a function call (13.10.2.1), 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 \( P \). [\texttt{Example}:

\begin{verbatim}
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 \}; // error: not a single element
auto x4 = \{ 3 \}; // decltype(x4) is std::initializer_list<int>
auto x5\{ 3 \}; // decltype(x5) is int
\end{verbatim}

—end \texttt{example}]"
template <class ...T> struct A {
    A(T...) {};
};

A x[29]{}; // error: no declarator operators allowed
const & 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 (12.4.1.8). 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.

2 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 (7.6.2.7), as the simple-type-specifier in an explicit type conversion (functional notation) (7.6.1.3), 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.

3 [Example:
    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 = { /* ... */ };
    container c(7); // 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

-- end example]

9.3 Declarators [dcl.decl]

1 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

    init-declarator:
      declarator initializer_opt
      declarator requires-clause

2 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.

3 Each init-declarator in a declaration is analyzed separately as if it was in a declaration by itself. [Note: 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

    struct S { /* ... */ };
    S S, T;  // declare two instances of struct S

which is not equivalent to

    struct S { /* ... */ };
    S S;

§ 9.3 174
Another exception is when \( T \) is \texttt{auto} (9.2.8.5), for example:

\begin{verbatim}
auto i = 1, j = 2.0; // error: deduced types for i and j do not match
\end{verbatim}
as opposed to

\begin{verbatim}
auto i = 1; // OK: i deduced to have type int
auto j = 2.0; // OK: j deduced to have type double
\end{verbatim}

— end note]  

The optional \texttt{requires-clause} (13.1) in an \texttt{init-declarator} or \texttt{member-declarator} shall be present only if the declarator declares a templated function (9.3.3.5). When present after a declarator, the \texttt{requires-clause} is called the \texttt{trailing requires-clause}. The trailing \texttt{requires-clause} introduces the \texttt{constraint-expression} that results from interpreting its \texttt{constraint-logical-or-expression} as a \texttt{constraint-expression}. [Example:

\begin{verbatim}
void f1(int a) requires true; // error: non-templated function
template<typename T>
  auto f2(T a) \to bool requires true; // OK
template<typename T>
  auto f3(T a) requires true \to bool; // error: requires-clause precedes trailing-return-type
void (*pf)() requires true; // error: constraint on a parameter-declaration
void g(int (*)() requires true); // error: constraint on a variable
auto* p = new void(*)(char) requires true; // error: not a function declaration
\end{verbatim}

— end example]

5 Declarators have the syntax

\begin{verbatim}
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\opt attribute-specifier-seq\opt

trailing-return-type:
  \to 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:
  \ldots \opt id-expression
\end{verbatim}
9.3.1 Type names

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\_opt parameters-and-qualifiers
  noptr-abstract-pack-declarator\_opt [ 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:

```
int i // int i
int *pi // int *pi
int *(*p3i)[3] // int (*p3i)[3]
int *f() // int *f()
int *(*pf)(double) // int (*pf)(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` (9.2.3).

9.3.2 Ambiguity resolution

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: 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:

```
struct S {
  S(int);
};

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
```
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:

```c
template <class T> struct X {};  
template <int N> struct Y {};  
X<int()> a;  // type-id  
X<int(1)> b;  // expression (ill-formed)  
Y<int()> c;  // type-id (ill-formed)  
Y<int(1)> 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:

```c
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,

```c
class C { };  
void h(int *(C[10]));  // void h(int *(fp)(C _parm[10]));  // not: void h(int *C[10]);
```

### 9.3.3 Meaning of declarators

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 (11.4.4, 11.4.7, 11.4.6, 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 (9.8.1)) 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: 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.
2 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.

3 Thus, a declaration of a particular identifier has the form

\[ T \rightarrow D \]

where \( T \) is of the form attrib-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.

4 First, the decl-specifier-seq determines a type. In a declaration

\[ T \rightarrow D \]

the decl-specifier-seq \( T \) determines the type \( T \). [Example: In the declaration

\[ int unsigned i; \]

the type specifiers \( int \) unsigned determine the type “unsigned int” (9.2.8.2). —end example]

5 In a declaration attrib-specifier-seq\_opt \( T \rightarrow D \) where \( D \) is an unadorned identifier the type of this identifier is “\( T \)”.

6 In a declaration \( T \rightarrow D \) where \( D \) has the form

\[ ( \rightarrow D1 ) \]

the type of the contained declarator-id is the same as that of the contained declarator-id in the declaration

\[ T \rightarrow D1 \]

Parentheses do not alter the type of the embedded declarator-id, but they can alter the binding of complex declarators.

9.3.3.1 Pointers

1 In a declaration \( T \rightarrow D \) where \( D \) has the form

\[ * attrib-specifier-seq\_opt cv-qualifier-seq\_opt D1 \]

and the type of the identifier in the declaration \( T \rightarrow D1 \) 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 attrib-specifier-seq (9.12.1) appertains to the pointer and not to the object pointed to.

2 [Example: The declarations

\[ const int ci = 10, *pc = &ci, *const cpc = pc, **ppc; \]

\[ int i, *p, *const cp = &i; \]

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

\[ i = ci; \]
\[ *cp = ci; \]
\[ pc++; \]
\[ pc = cpc; \]
\[ pc = p; \]
\[ ppc = &pc; \]

Examples of ill-formed operations are

\[ ci = 1; \]
\[ ci++; \]
\[ *pc = 2; \]
\[ cp = &ci; \]
\[ cpc++; \]
\[ p = pc; \]
\[ ppc = &p; \]

Each is unacceptable because it would either change the value of an object declared 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

§ 9.3.3.1 178
\*p = 5; // clobber ci

— end example

3 See also 7.6.19 and 9.4.

4 [Note: Forming a pointer to reference type is ill-formed; see 9.3.3.2. Forming a function pointer type is ill-formed if the function type has cv-qualifiers or a ref-qualifier; see 9.3.3.5. Since the address of a bit-field (11.4.9) cannot be taken, a pointer can never point to a bit-field. — end note]

9.3.3.2 References [dcl.ref]

1 In a declaration \(T D\) where \(D\) has either of the forms

\[
& \text{attribute-specifier-seq}_{\text{opt}} D1
&& \text{attribute-specifier-seq}_{\text{opt}} D1
\]

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 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 (9.2.3, 13.2) or decltype-specifier (9.2.8.2), in which case the cv-qualifiers are ignored. [Example:

```c
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: 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:

```c
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 \(d\).

```c
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,

```c
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);
}
```

declares \(p\) to be a reference to a pointer to \(link\) so \(h(q)\) will leave \(q\) with the value zero. See also 9.4.3. — end example]
It is unspecified whether or not a reference requires storage (6.7.5).

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 (9.4.3) except when the declaration contains an explicit extern specifier (9.2.1), is a class member (11.4) declaration within a class definition, or is the declaration of a parameter or a return type (9.3.3.5); see 6.2. A reference shall be initialized to refer to a valid object or function. [Note: 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 11.4.9, a reference cannot be bound directly to a bit-field. —end note]

If a typedef-name (9.2.3, 13.2) or a decltype-specifier (9.2.8.2) 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 “rvalue reference to cv TR” creates the type TR. [Note: This rule is known as reference collapsing. —end note] [Example:

```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 &&

decatype(r2)& r6 = i; // r6 has the type int &
decatype(r2)&& r7 = i; // r7 has the type int &&
```

—end example]

[Note: Forming a reference to function type is ill-formed if the function type has cv-qualifiers or a ref-qualifier; see 9.3.3.5. —end note]

9.3.3.3 Pointers to members [decl.mptr]

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 T”. The optional attribute-specifier-seq (9.12.1) appertains to the pointer-to-member.

2 [Example:

```c
struct X {
    void f(int);
    int a;
};

struct Y;

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares pmi, pmf, pmd and pmc 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 pmc is well-formed even though Y is an incomplete type. pmi 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
```
A pointer to member shall not point to a static member of a class (11.4.8), a member with reference type, or “cv void”.

[Note: 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]

9.3.3.4 Arrays

In a declaration T D where D has the form

\[ D1 \text{[constant-expression\_opt \ attribute-specifier-seq\_opt} \]

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 array of N T”. The constant-expression shall be a converted constant expression of type 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.

In a declaration T D where D has the form

\[ D1 \text{[ attribute-specifier-seq\_opt} \]

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 array of unknown bound of T", except as specified below.

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.

U is called the array element type; this type shall not be a placeholder type (9.2.8.5), a reference type, a function type, an array of unknown bound, or cv void. [Note: 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 type bound. — end note] [Example:

\[
\text{float fa[17], *afp[17];}
\]
declares an array of float numbers and an array of pointers to float numbers. — end example]

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:

\[
\text{typedef int A[5], AA[2][3];}
\]
\[
\text{typedef const A CA; \quad \text{// type is “array of 5 const int”}}
\]
\[
\text{typedef const AA CAA; \quad \text{// type is “array of 2 array of 3 const int”}}
\]
— end example] [Note: An “array of N cv-qualifier-seq U” has cv-qualified type; see 6.8.3. — end note]

An object of type “array of N U” contains 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.

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 (9.3.3.5). 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, 7.6.1.3, 7.6.2.7). In these cases, the array bound is calculated from the number of initial elements (say, N) supplied (9.4.1), and the type of the array is “array of N U”.

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:

\[
\text{extern int x[10];}
\]
\[
\text{struct S {}
\quad \text{static int y[10];}
\};}
\]
\[
\text{int x[]; \quad \text{// OK: bound is 10}}
\]
\[
\text{int S::y[]; \quad \text{// OK: bound is 10}}
\]
\[
\text{void f()} \{
\quad \text{extern int x[];}
\}
\]
In a declaration $T D$ where $D$ has the form

$$D1 (\text{parameter-declaration-clause}) \text{cv-qualifier-seq}\text{opt} \quad \text{ref-qualifier}\text{opt noexcept-specifier}\text{opt attribute-specifier-seq}\text{opt}$$

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 noexcept-specifier\text{opt} function of parameter-type-list\text{cv-qualifier-seq}\text{opt} ref-qualifier\text{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.

In a declaration $T D$ where $D$ has the form

$$D1 (\text{parameter-declaration-clause}) \text{cv-qualifier-seq}\text{opt} \quad \text{ref-qualifier}\text{opt noexcept-specifier}\text{opt attribute-specifier-seq}\text{opt} trailing-return-type$$

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\text{opt} function of parameter-type-list\text{cv-qualifier-seq}\text{opt} ref-qualifier\text{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.\textsuperscript{87}

\text{parameter-declaration-clause:}
\text{parameter-declaration-list}\text{opt} \cdots \text{opt}
\text{parameter-declaration-list} , \ldots

\text{parameter-declaration-list:}
\text{parameter-declaration}
\text{parameter-declaration-list} , \text{parameter-declaration}

\textsuperscript{87} As indicated by syntax, cv-qualifiers are a significant component in function return types.
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: The parameter-declaration-clause is used to convert the arguments specified on the function call; see 7.6.1.2. —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.5. If the parameter-declaration-clause terminates with an ellipsis or a function parameter pack (13.7.3), 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: The declaration

```c
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```c
printf("hello world");
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]

[Note: The standard header <cstdio> (17.13.1) contains a mechanism for accessing arguments passed using the ellipsis (see 7.6.1.2 and 17.13). —end note]

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: This transformation does not affect the types of the parameters. For example, int(*)(const int p, decltype(p)*) and 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 (9.2.3, 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 (13.4.1).

[Example:

```c
typedef int FIC(int) const;
FIC f; // error: does not declare a member function
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: 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:

```c
typedef void F();
struct S {
  const F f; // OK: equivalent to: void f();
};
```
The return type, the parameter-type-list, the ref-qualifier, the cv-qualifier-seq, and the exception specification, but not the default arguments (9.3.3.6) or the trailing requires-clause (9.3), are part of the function type. [Note: Function types are checked during the assignments and initializations of pointers to functions, references to functions, and pointers to member functions. — end note]

Example: The declaration

```c
int fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types, and returning int (9.2.8). — 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 (13.7.6.1).

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.5.

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:

```c
typedef void F();
F fv;
// OK: equivalent to void fv();
F fv {} // error
void fv {} // OK: definition of fv
```

An identifier can optionally be provided as a parameter name; if present in a function definition (9.5), it names a parameter. [Note: 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: The declaration

```c
int i,
  *pi,
  f(),
  *fpi(int),
  (*pif)(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 fpi and pif. The binding of *fpi(int) is *(fpi(int)), so the declaration suggests, and the same construction in an expression requires, the calling of a function fpi, and then using indirection through the (pointer) result to yield an integer. In the declarator (*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: Typedefs and trailing-return-types are sometimes convenient when the return type of a function is complex. For example, the function fpif above could have been declared

```c
typedef int IFUNC(int);
IFUNC* fpif(int);
```
or

```c
auto fpif(int)->int(*)(int);
```

A trailing-return-type is most useful for a type that would be more complicated to specify before the declarator-id:

```c
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```
rather than

```
    template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

—end note

17 A **non-template function** is a function that is not a function template specialization. [Note: A function template is not a function. —end note]

18 An **abbreviated function template** is a function declaration that has one or more generic parameter type placeholders (9.2.8.5). An abbreviated function template is equivalent to a function template (13.7.6) 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 `auto`, the invented parameter is an unconstrained **type-parameter**. For a **placeholder-type-specifier** of the form `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 (9.3.3.5). If the placeholder contains `decltype(auto)`, the program is ill-formed. The adjusted function parameters of an abbreviated function template are derived from the **parameter-declaration-clause** by replacing each occurrence of a placeholder with the name of the corresponding invented **template-parameter**. [Example:

```
    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);

These declarations are functionally equivalent (but not equivalent) to the following declarations.

```
    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);
```

Abbreviated function templates can be specialized like all function templates.

```
    template<> void g1<int>(const int*, int<double&>);
```

—end example

19 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:

```
    template<typename> concept C = /* ... */;
```

```
    template <typename T, C U>
    void g(T x, U y, C auto z);
```

This is functionally equivalent to each of the following two declarations.

```
    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 (13.7.3). 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 (13.7.3). [Example:

```
    template<typename... T> void f(T (*...t)(int, int));
```

```
    int add(int, int);
```

§ 9.3.3.5 185
float subtract(int, int);

void g() {
    f(add, subtract);
}
—end example]

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.  

9.3.3.6 Default arguments [decl.fct.default]

1 If an initializer-clause is specified in a parameter-declaration this initializer-clause is used as a default argument. [Note: Default arguments will be used in calls where trailing arguments are missing (7.6.1.2). — end note]

2 [Example: 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]

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: A default argument cannot be redefined by a later declaration (not even to the same value) (6.3). — end note] [Example:

    void g(int = 0, ...);
    // OK, ellipsis is not a parameter so it can follow
    // a parameter with a default argument

    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);
        void f(int, int = 5);
        // OK
        f(4);
        // OK, calls f(4, 5);
        void 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...);
    };

88) 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).

89) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or typedef declarations.
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 13.9.1. [Example: In the following code, \texttt{g} will be called with the value \texttt{f(2)}:

```c
int a = 1;
int f(int);
ing(int x = f(a)); // default argument: \texttt{f(:a)}
void h() {
  a = 2;
  {
    int a = 3;
    g(); // g(f(:a))
  }
}
```

—end example] [Note: In member function declarations, names in default arguments are looked up as described in 6.5.1. Access checking applies to names in default arguments as described in 11.9. —end note]

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 (11.4.4.1), copy or move constructor (11.4.4.2), or copy or move assignment operator (11.4.5) 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:

```c
class C {
  void f(int i = 3);
  void g(int i, int j = 99);
};
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: A local variable cannot be odr-used (6.3) in a default argument. —end note] [Example:

```c
void f() {
  int i;
  extern void g(int x = i); // error
  extern void h(int x = sizeof(i)); // OK
  // ...
}
```

—end example]

[Note: The keyword \texttt{this} may not appear in a default argument of a member function; see 7.5.2. [Example:

```c
class A {
  void f(A* p = this) {} // error
};
```

—end example] —end note]

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:
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 (7.6.1.4) or unless it is used to form a pointer to member (7.6.2.1). [Example: 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:

```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

—end example— 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.

10 A virtual function call (11.7.2) 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:

```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()
}
—end example—

9.4 Initializers [dcl.init]

1 The process of initialization described in this subclause applies to all initializations regardless of syntactic context, including the initialization of a function parameter (7.6.1.2), the initialization of a return value (8.7.3), or when an initializer follows a declarator.

initializer:
brace-or-equal-initializer
( expression-list )

§ 9.4 188
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

[Note: The rules in this subclause apply even if the grammar permits only the brace-or-equal-initializer form of initializer in a given context. — end note]

2 Except for objects declared with the constexpr specifier, for which see 9.2.5, 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:

```
int f(int);
int a = 2;
int b = f(a);
int c(b);
```
— end example]

3 [Note: Default arguments are more restricted; see 9.3.3.6. — end note]

4 [Note: The order of initialization of variables with static storage duration is described in 6.9.3 and 8.8. — end note]

5 A declaration of a block-scope variable with external or internal linkage that has an initializer is ill-formed.

6 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.

7 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 (12.4.1.3), and the best one for the initializer 0 is chosen through overload

---

90) As specified in 7.3.11, converting an integer literal whose value is 0 to a pointer type results in a null pointer value.
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 \textit{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.

8 To \textit{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 (11.4.4.1) or a default constructor that is user-provided or deleted, then the object is default-initialzed;
(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.

9 A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

[Note: 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]

10 An object whose initializer is an empty set of parentheses, i.e., \( () \), shall be value-initialized.

[Note: Since \( () \) is not permitted by the syntax for \textit{initializer},
\[
\text{X a();}
\]

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 (7.6.2.7, 7.6.1.3, 11.10.2).
— end note]

11 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:
\[
\text{int a;}
\]

\[
\text{struct X {
    static int a;
    static int b;
};}
\]

\[
\text{int X::a = 1;}
\]

\[
\text{int X::b = a; // X::b = X::a}
\]

— end example]

12 If the entity being initialized does not have class type, the \textit{expression-list} in a parenthesized initializer shall be a single expression.

13 The initialization that occurs in the \( = \) form of a \textit{brace-or-equal-initializer} or \textit{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 (9.4.1), is called copy-initialization. [Note: Copy-initialization may invoke a move (11.4.4.2). — end note]

The initialization that occurs

- for an initializer that is a parenthesized expression-list or a braced-init-list,
- for a new-initializer (7.6.2.7),
- in a static_cast expression (7.6.1.8),
- in a functional notation type conversion (7.6.1.3), 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 (9.4.4).

- If the destination type is a reference type, see 9.4.3.

- 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 9.4.2.

- If the initializer is 0, the object is value-initialized.

- 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: \( T x = T(T(T())) \); 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 (12.4.1.3), 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 (9.4.1). 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: By contrast with direct-list-initialization, narrowing conversions (9.4.4) 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:]

```cpp
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 12.4.1.4, 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 (12.4.1.5), 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 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: An expression of type “cv1 T” can initialize an object of type “cv2 T” independently of the cv-qualifiers cv1 and cv2.

---

18 An initializer-clause followed by an ellipsis is a pack expansion (13.7.3).

19 If the initializer is a parenthesized expression-list, the expressions are evaluated in the order specified for function calls (7.6.1.2).

20 The same identifier shall not appear in multiple designators of a designated-initializer-list.

21 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: Such an object might have been value-initialized or initialized by aggregate initialization (9.4.1) or by an inherited constructor (11.10.3). — 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.

22 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: In most cases this is the defining declaration (6.2) of the variable, but the initializing declaration of a non-inline static data member (11.4.8.2) might be the declaration within the class definition and not the definition at namespace scope. — end note]

9.4.1 Aggregates

An aggregate is an array or a class (Clause 11) with

(1.1) — no user-declared or inherited constructors (11.4.4),
(1.2) — no private or protected direct non-static data members (11.9),

§ 9.4.1 192
1.3 — no virtual functions (11.7.2), and

1.4 — no virtual, private, or protected base classes (11.7.1).

[Note: 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 9.4.4, 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:

```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 (9.4.4) is required to convert the expression, the program is ill-formed. [Note: 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:

```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;
};
```
For a non-union aggregate, each element that is not an explicitly initialized element is initialized as follows:

1. If the element has a default member initializer (11.4), the element is initialized from that initializer.
2. Otherwise, if the element is not a reference, the element is copy-initialized from an empty initializer list (9.4.4).
3. Otherwise, the program is ill-formed.

If the aggregate is a union and the initializer list is empty, then

1. If any variant member has a default member initializer, that member is initialized from its default member initializer;
2. otherwise, the first member of the union (if any) is copy-initialized from an empty initializer list.

Example:
```cpp
struct S { int a; const char* b; int c; int d = b[a]; }; 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
```

```cpp
struct X { int i, j, k = 42; }; X a[1] = { 1, 2, 3, 4, 5, 6 }; X b[2] = { { 1, 2, 3 }, { 4, 5, 6 } };
a and b have the same value
```

```cpp
struct A {
  string a;
  int b = 42;
  int c = -1;
}; A{.c=21} has the following steps:
```

1. Initialize a with {}
2. Initialize b with = 42
3. Initialize c with = 21
   
   —end example}]

6. 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.

7. An aggregate that is a class can also be class initialized with a single expression not enclosed in braces, as described in 9.4.

8. The destructor for each element of class type is potentially invoked (11.4.6) from the context where the aggregate initialization occurs. [Note: This provision ensures that destructors can be called for fully-constructed subobjects in case an exception is thrown (14.3). —end note]

9. An array of unknown bound initialized with a brace-enclosed `initializer-list` containing n `initializer-classes` is defined as having n elements (9.3.3.4). [Example:
```cpp
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: 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 (11.10.2), the default member initializer is not considered to initialize the array of unknown bound. [Example:

91) The syntax provides for empty `braced-init-lists`, but nonetheless C++ does not have zero length arrays.

§ 9.4.1
struct S {
    int y[] = {0}; // error: non-static data member of incomplete type
};
—end example] — end note]

[Note: Static data members, non-static data members of anonymous union members, and unnamed bit-fields are not considered elements of the aggregate. [Example:

struct A {
    int i;
    static int s;
    int j;
    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] — end note]

An initializer-list is ill-formed if the number of initializer-clauses exceeds the number of elements of the aggregate. [Example:

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:

struct A;
extern A a;
struct A {
    const & a1 { A{a,a} }; // OK
    const & a2 { A{} }; // error
};
A a(a,a);
// OK

struct B {
    int n = B{}.n; // error
};
—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:

struct S { } s;
struct A {
    S s1;
    int i1;
    S s2;
    int i2;
    S s3;
    int i3;
} a = {
    { }, // Required initialization
    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 (9.3.3.4). [Example:

int x[2][2] = {3, 1, 4, 2};
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 }
};

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:

```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]

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: As specified above, brace elision cannot apply to subaggregates with no elements; an initializer-clause for the entire subobject is required. — end note]

[Example:

```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]

[Note: An aggregate array or an aggregate class may contain elements of a class type with a user-declared constructor (11.4.4). Initialization of these aggregate objects is described in 11.10.1. — end note]

[Note: 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]

When a union is initialized with an initializer list, there shall not be more than one explicitly initialized element. [Example:

```c
union u { int a; const char* b; };
u a = { 1 };
u b = a;
u c = 1;
u d = { 0, "asdf" };  // error
u e = { "asdf" };      // error
```

§ 9.4.1
u f = { .b = "asdf" };  
// error
— end example]

Note: 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]

9.4.2 Character arrays  

An array of ordinary character type (6.8.1), `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:  

```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]

There shall not be more initializers than there are array elements. [Example:  

```c
char cv[4] = "asdf";  
// error
```

is ill-formed since there is no space for the implied trailing `\0`. — end example]

If there are fewer initializers than there are array elements, each element not explicitly initialized shall be zero-initialized (9.4).

9.4.3 References  

A variable whose declared type is “reference to type T” (9.3.3.2) shall be initialized. [Example:  

```c
void f() {
  int i;
  int& r = i;  
  // r refers to i
  r = i;  
  // 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 (a[r])[3] = a;  
  // ra refers to the array a
  ra[1] = i;  
  // modifies a[1]
}
```

— end example]

A reference cannot be changed to refer to another object after initialization. [Note: Assignment to a reference assigns to the object referred to by the reference (7.6.19). — end note] Argument passing (7.6.1.2) and function value return (8.7.3) are initializations.

The initializer can be omitted for a reference only in a parameter declaration (9.3.3.5), 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:  

```c
int& r1;  
// error: initializer missing
extern int& r2;  
// OK
```

— end example]

Given types “cv1 T1” and “cv2 T2”, “cv1 T1” is reference-related to “cv2 T2” if T1 is similar (7.3.5) 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 cv2 T2” 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.

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
— is an lvalue (but is not a bit-field), and “cv1 T1” is reference-compatible with “cv2 T2”, or

— 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”\(^\text{92}\) (this conversion is selected by enumerating the applicable conversion functions (12.4.1.6) 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: The usual lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. — end note]

[Example:

```cpp
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 A& 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:

```cpp
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

— is an rvalue (but not a bit-field) or function lvalue and “cv1 T1” is reference-compatible with “cv2 T2”, or

— 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 lvalue of type “cv3 T3”, where “cv1 T1” is reference-compatible with “cv3 T3” (see 12.4.1.6),

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” (7.3.5) and the temporary materialization conversion (7.3.4) is applied. In any case, the reference is bound to the resulting glvalue (or to an appropriate base class subobject).

[Example:

```cpp
struct A { }
struct B : A { } b;
extern B f();
const & rca2 = f();  // bound to the A subobject of the B rvalue.
& & 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& rri = static_cast<int&>(i2);  // bound directly to i2
& & rrb = x;  // bound directly to the result of operator B
```
— end example]

(5.4) — Otherwise:

\(^{92}\) This requires a conversion function (11.4.7.2) returning a reference type.
If $T_1$ or $T_2$ is a class type and $T_1$ is not reference-related to $T_2$, user-defined conversions are considered using the rules for copy-initialization of an object of type “cv1 $T_1$” by user-defined conversion (9.4, 12.4.1.4, 12.4.1.5); 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 $T_1$”. The temporary materialization conversion is applied and the reference is bound to the result.

If $T_1$ is reference-related to $T_2$:

- 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:

```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;
struct A { 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
```

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: 6.7.7 describes the lifetime of temporaries bound to references. — end note]

9.4.4 List-initialization

List-initialization is initialization of an object or reference from a braced-init-list. Such an initializer can be called an initializer list, and the comma-separated initializer-clauses or designated-initializer-clauses of the initializer-list or 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. [Note: List-initialization can be used

1 as the initializer in a variable definition (9.4)
2 as the initializer in a new-expression (7.6.2.7)
3 in a return statement (8.7.3)
4 as a for-range-initializer (8.6)
5 as a function argument (7.6.1.2)
6 as a subscript (7.6.1.1)
7 as an argument to a constructor invocation (9.4, 7.6.1.3)
— as an initializer for a non-static data member (11.4)
— in a mem-initializer (11.10.2)
— on the right-hand side of an assignment (7.6.19)

[Example:
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"};  // 4 string elements
f ({"Nicholas","Annemarie"});  // pass list of two elements
return { "Norah" };  // return list of one element
int* e {};
// initialization to zero / null pointer
x = double{1};  // explicitly construct a double
std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };

—end example]  —end note]

2 A constructor is an initializer-list constructor if its first parameter is of type std::initializer_list<E> or reference to cv std::initializer_list<E> for some type E, and either there are no other parameters or else all other parameters have default arguments (9.3.3.6). [Note: Initializer-list constructors are favored over other constructors in list-initialization (12.4.1.7). Passing an initializer list as the argument to the constructor template template<class T> C(T) of a class C does not create an initializer-list constructor, because an initializer list argument causes the corresponding parameter to be a non-deduced context (13.10.2.1). — end note] The template std::initializer_list is not predefined; if the header <initializer_list> is not imported or included prior to a use of std::initializer_list — even an implicit use in which the type is not named (9.2.8.5) — the program is ill-formed.

3 List-initialization of an object or reference of type T is defined as follows:

(3.1) — If the braced-init-list contains a designated-initializer-list, T shall be an aggregate class. The ordered identifiers in the designators of the designated-initializer-list shall form a subsequence of the ordered identifiers in the direct non-static data members of T. Aggregate initialization is performed (9.4.1). [Example:
struct A { int x; int y; int z; };
A a{.y = 2, .x = 1};  // error: designator order does not match declaration order
A b{.x = 1, .z = 2};  // OK, b.y initialized to 0
—end example]

(3.2) — If T is an aggregate class and the initializer list has a single element of type cv U, where U is T or a class derived from T, the object is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization).

(3.3) — Otherwise, if T is a character array and the initializer list has a single element that is an appropriately-typed string-literal (9.4.2), initialization is performed as described in that subclause.

(3.4) — Otherwise, if T is an aggregate, aggregate initialization is performed (9.4.1).

[Example:
double ad[] = { 1, 2.0 };  // OK
int ai[] = { 1, 2.0 };  // error: narrowing

struct S2 {
int m1;
double m2, m3;
};
S2 s21 = { 1, 2, 3.0 };  // OK
S2 s22 { 1.0, 2, 3 };  // error: narrowing
S2 s23 { };  // OK: default to 0,0,0
—end example]

(3.5) — Otherwise, if the initializer list has no elements and T is a class type with a default constructor, the object is value-initialized.

(3.6) — Otherwise, if T is a specialization of std::initializer_list<E>, the object is constructed as described below.

§ 9.4.4 200
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, 12.4.1.7). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

**Example:**

```cpp
struct S {
    S(std::initializer_list<double>); // #1
    S(std::initializer_list<int>); // #2
    S(); // #3
    // ...
};
S s1 = { 1.0, 2.0, 3.0 }; // invoke #1
S s2 = { 1, 2, 3 }; // invoke #2
S s3 = {}; // invoke #3
```

— end example]

**Example:**

```cpp
struct Map {
    Map(std::initializer_list<std::pair<std::string,int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};
```

— end example]

**Example:**

```cpp
struct S {
    S(int, double, double); // #1
    S(); // #2
    // ...
};
S s1 = { 1, 2, 3.0 }; // OK: invoke #1
S s2 { 1.0, 2, 3 }; // error: narrowing
S s3 {}; // OK: invoke #2
```

— end example]

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 \( T(v) \) (7.6.1.3); if a narrowing conversion is required to convert \( v \) to \( U \), the program is ill-formed. **Example:**

```cpp
enum byte : unsigned char { }; byte b { 42 }; // OK
byte c = { 42 }; // error
byte d = byte{ 42 }; // OK; same value as b
byte e { -1 }; // error
```

```cpp
struct A { byte b; }
A a1 = { { 42 } }; // error
A a2 = { byte{ 42 } }; // OK
```

```cpp
void f(byte);
f({ 42 }); // error
```

```cpp
enum class Handle : uint32_t { Invalid = 0 }
Handle h { 42 }; // OK
```

— end example]

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:**

```cpp
§ 9.4.4 201
```
int x1 {2};  // OK
int x2 {2.0};  // error: narrowing
—end example[3.10]
—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: 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:

```cpp
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
```
—end example[3.11]
—Otherwise, if the initializer list has no elements, the object is value-initialized.

[Example:

```cpp
int** pp {}; // initialized to null pointer
—end example
```
—end example[3.12]
—Otherwise, the program is ill-formed.

[Example:

```cpp
struct A { int i; int j; }; // aggregate initialization
struct B (std::initializer_list<int>); // creates std::initializer_list<int> and calls constructor
struct C (int 1, double j); // calls constructor with arguments (1, 2.2)
struct D { explicit B(const A&); // error: cannot copy-list-initialize B temporary from A
```
—end example[4]

---

4 Within the `initializer-list` of a `braced-init-list`, the `initializer-clauses`, including any that result from pack expansions (13.7.3), 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: 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 (7.3.4) 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: 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:

```c++
struct X {
    X(std::initializer_list<double> v);
}
X x{ 1,2,3 };
```

The initialization will be implemented in a way roughly equivalent to this:

```c++
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:

```c++
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 (11.10.2). — end example] [Note: 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

- from a floating-point type to an integer type, or
- 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
- 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
- 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
- from a pointer type or a pointer-to-member type to `bool`.

[Note: As indicated above, such conversions are not allowed at the top level in list-initializations. — end note] [Example:

```c++
int x = 999;
// x is not a constant expression
const int y = 999;
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

```
function-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 9.3.3.5. 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: 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]

A ctor-initializer is used only in a constructor; see 11.4.4 and 11.10.

[Note: A cv-qualifier-seq affects the type of this in the body of a member function; see 9.3.3.2. — end note]

[Note: Unused parameters need not be named. For example,

```
void print(int a, int) {
    std::printf("a = %d\n", a);
}
```

— end note]

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).

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 \textit{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.\textsuperscript{93} \textbf{[Example:}

\begin{verbatim}
struct S {
    S() : s(__func__) { } // OK
    const char* s;
};
void f(const char* s = __func__); // error: __func__ is undeclared
— end example]
\end{verbatim}

9.5.2 Explicitly-defaulted functions \hfill \textbf{[dcl.fct.def.default]}\textsuperscript{1}

A function definition whose \textit{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 (12.6.2), and
\item not have default arguments.
\end{enumerate}

\begin{enumerate}
\item 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 constexpr-compatible (11.4.3, 11.11.1). A function explicitly defaulted on its first declaration is implicitly inline (9.2.7), and is implicitly constexpr (9.2.5) if it is constexpr-compatible.

\begin{enumerate}
\item \textbf{[Example:}

\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;
};
U u1;
U u2 = static_cast<U&&>(u1); // OK, calls std::terminate if \texttt{T::T(T&&)} throws
— end example]
\end{verbatim}

\textsuperscript{93} 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.

\section*{§ 9.5.2 205}
Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (11.4.4, 11.4.6, 11.4.4.2, 11.4.5), which might mean defining them as deleted. A defaulted prospective destructor (11.4.6) 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 (11.4.3) 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: 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:

```cpp
struct trivial {
    trivial() = default;
    trivial(const trivial&) = default;
    trivial(trivial&&) = default;
    trivial& operator=(const trivial&) = default;
    -trivial() = default;
};

struct nontrivial1 {
    nontrivial1();
};
nontrivial1::nontrivial1() = default; // not first declaration
```
— end example]

9.5.3 Deleted definitions [dcl.fct.def.delete]

1 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*.

2 A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [Note: 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]

3 [Example: 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: 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: 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

A deleted function is implicitly an inline function (9.2.7). [Note: 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 (6.7.5.4) shall not be defined as deleted. [Example:

```cpp
struct sometype {
    sometype();
};
sometype::sometype() = delete;  // error: not first declaration
```

—end example]

### 9.5.4 Coroutine definitions [dcl.fct.def.coroutine]

A function is a coroutine if its function-body encloses a coroutine-return-statement (8.7.4), an await-expression 
(7.6.2.3), 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.

[Example:

```cpp
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]

The promise type of a coroutine is `std::coroutine_traits<R, P₁, ..., Pₙ>::promise_type`, where R is 
the return type of the function, and P₁...Pₙ are the sequence of types of the function parameters, preceded 
by the type of the implicit object parameter (12.4.1) if the coroutine is a non-static member function. The 
promise type shall be a class type.

In the following, pᵢ is an lvalue of type Pᵢ, where p₁ denotes *this and pᵢ₊₁ denotes the ℐᵗʰ function parameter 
for a non-static member function, and pᵢ denotes the ℐᵗʰ function parameter otherwise.

A coroutine behaves as if its function-body were replaced by:
{  
  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 (7.6.2.3) 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 (8.7.4), 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 (12.4.2), 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: 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 (8.7.4). — 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
   (17.12.3.4) of a coroutine handle (17.12.3) 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 (6.7.5.4.1). 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 (12.4.2), 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 (6.5.5). 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, nothrow_t) form
    is used. The allocation function used in this case shall have a non-throwing noexcept-specification. 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:
```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() { 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 (17.12.3.4) of a coroutine handle (17.12.3) 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 (6.7.5.4.2). 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 (7.6.1.2), 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: An original parameter object is never a const or volatile object (6.8.3). — 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: 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]
If the evaluation of the expression `promise.unhandled_exception()` exits via an exception, the coroutine is considered suspended at the final suspend point.

The expression `co_wait promise.final_suspend()` shall not be potentially-throwing (14.5).

### 9.6 Structured binding declarations

A structured binding declaration introduces the identifiers \(v_0, v_1, v_2, \ldots\) of the identifier-list as names (6.4.1) of structured bindings. Let `cv` denote the cv-qualifiers in the `decl-specifier-seq` and `S` consist of the storage-class-specifiers of the `decl-specifier-seq` (if any). A `cv` that includes `volatile` is deprecated; see D.5. 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
class-specifier cv
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: \(E\) is never a reference type (7.2). —end note]

If the `initializer` refers to one of the names introduced by the structured binding declaration, the program is ill-formed.

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 `v_i` 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: The top-level `cv-qualifiers` of `T` are `cv`. —end note] [Example:

```cpp
auto f() -> int(&)[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]
```

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 prvalue of type `std::size_t` corresponding to `v_i`. The unqualified-id `get` is looked up in the scope of `E` by class member access lookup (6.5.5), 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 (6.5.2). In either case, `get\(<i>` is interpreted as a template-id. [Note: Ordinary unqualified lookup (6.5.1) 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 `T_i` designated by `std::tuple_element<i, E>::type` and the type `U_i` designated by either `T_i&` or `T_i&&`, where `U_i` is an lvalue reference if the initializer is an lvalue and an xvalue reference otherwise, variables are introduced with unique names `r_i` as follows:

```cpp
S U_i, r_i = initializer ;
```

Each `v_i` is the name of an lvalue of type `T_i` that refers to the object bound to `r_i`; the referenced type is `T_i`.

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 `m_0, m_1, m_2, ...` (in declaration order), each `v_i` is the name of an lvalue that refers to the member `m_i` of `e` and whose type is `cv T_i`, where `T_i` is the declared type of that member; the referenced type is `cv T_i`. The lvalue is a bit-field if that member is a bit-field. [Example:

```cpp
struct 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

9.7.1 Enumeration declarations

An enumeration is a distinct type (6.8.2) 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: 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:

```
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 (13.9.3).

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 *type-specifier-seq* of an *enum-base* shall name an integral type; any cv-qualification is ignored. An *opaque-enum-declaration* declaring an unscoped enumeration shall not omit the *enum-base*. The identifiers in an *enumerator-list* are declared as constants, and can appear wherever constants are required. An *enumerator-definition* with = gives the associated *enumerator* the value indicated by the *constant-expression*. If the first *enumerator* has no initializer, the value of the corresponding constant is zero. An *enumerator-definition* without an initializer gives the *enumerator* the value obtained by increasing the value of the previous *enumerator* by one. [Example:

```
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```

§ 9.7.1
defines \(a, c, \) and \(d\) to be zero, \(b\) and \(e\) to be 1, and \(f\) to be 3. —end example] The optional attribute-specifier-seq in an enumerator appertains to that enumerator.

3 An opaque-enum-declaration is either a redeclaration of an enumeration in the current scope or a declaration of a new enumeration. [Note: An enumeration declared by an 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 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 enum-base specifying the same underlying type as in the original declaration.

4 If an enum-head-name contains a nested-name-specifier, it shall not begin with a decltype-specifier and the enclosing enum-specifier or opaque-enum-declaration shall refer to an enumeration 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 (9.8.1) of that namespace (i.e., neither inherited nor introduced by a using-declaration), and the enum-specifier or opaque-enum-declaration shall appear in a namespace enclosing the previous declaration.

5 Each enumeration defines a type that is different from all other types. Each enumeration also has an underlying type. The underlying type can be explicitly specified using an enum-base. For a scoped enumeration type, the underlying type is int if it is not explicitly specified. In both of these cases, the underlying type is said to be fixed. Following the closing brace of an enum-specifier, each enumerator has the type of its enumeration. If the underlying type is fixed, the type of each enumerator prior to the closing brace is the underlying type and the constant-expression in the 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 enumerator prior to the closing brace is determined as follows:

\[(5.1)\] If an initializer is specified for an enumerator, the constant-expression shall be an integral constant expression (7.7). If the expression has unscoped enumeration type, the enumerator has the underlying type of that enumeration type, otherwise it has the same type as the expression.

\[(5.2)\] If no initializer is specified for the first enumerator, its type is an unspecified signed integral type.

\[(5.3)\] Otherwise the type of the enumerator is the same as that of the preceding 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.

6 An enumeration whose underlying type is fixed is an incomplete type from its point of declaration (6.4.2) to immediately after its enum-base (if any), at which point it becomes a complete type. An enumeration whose underlying type is not fixed is an incomplete type from its point of declaration to immediately after the closing } of its enum-specifier, at which point it becomes a complete type.

7 For an enumeration whose underlying type is not fixed, the underlying type is an integral type that can represent all the enumerator values defined in the enumeration. If no integral type can represent all the 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 int unless the value of an enumerator cannot fit in an int or unsigned int. If the enumerator-list is empty, the underlying type is as if the enumeration had a single enumerator with value 0.

8 For an enumeration 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 integer type with minimal width \(M\) 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 \(M\). It is possible to define an enumeration that has values not defined by any of its enumerators. If the enumerator-list is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.\(^{94}\)

9 Two enumeration types are layout-compatible enumerations if they have the same underlying type.

10 The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (7.3.6). [Example:

```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 };  // error: no Col to int conversion
int x = Col::red;  // error: no Col to int conversion
Col y = Col::red;  // 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 (9.2.3) 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 `enumerator` as a name for linkage purposes. These names obey the scope rules defined for all names in 6.4 and 6.5. [Note: Each unnamed enumeration with no enumerators is a distinct type. — end note] [Example:

```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
}
```

An `enumerator` declared in class scope can be referred to using the class member access operators (`::`, `. (dot) and -> (arrow)`, see 7.6.1.4. [Example:

```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
    // ...
}
```

---

9.7.2 The `using enum declaration` [enum.udecl]

```c
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: 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:

```cpp
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: Two using-enum-declarations that introduce two enumerators of the same name conflict. [Example:

```cpp
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

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: A namespace name with external linkage is exported if any of its namespace-declarations 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:

```cpp
export module M;
namespace N1 {}                  // N1 is not exported
export namespace N2 {}          // N2 is exported
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.

9.8.1 Namespace definition

```cpp
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-name-space-specifier :: inline_opt identifier { namespace-body }
enclosing-name-space-specifier:
    identifier
    enclosing-name-space-specifier :: inline_opt identifier
namespace-body:
    declaration-seq_opt
```

1 Every namespace-definition shall appear at namespace scope (6.4.6).
In a named-namespace-definition, the identifier is the name of the namespace. If the identifier, when looked up (6.5.1), refers to a namespace-name (but not a namespace-alias) that was introduced in the namespace in which the named-namespace-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-namespace-definition appears.

Because a namespace-definition contains declarations in its namespace-body and a namespace-definition is itself a declaration, it follows that namespace-definitions can be nested. [Example:

```cpp
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; } // Outer::i
    int i;
    void g() { i++; } // Inner::i
  }
}
```
—end example]

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 9.8.1.2). Such a redeclaration has the same enclosing namespaces as the original declaration. [Example:

```cpp
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
  }
  void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```
—end example]

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-namespace-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 (6.5.2) whenever one of them is, and a using-directive (9.8.3) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (9.8.1.1). Furthermore, each member of the inline namespace can subsequently be partially specialized (13.7.5), explicitly instantiated (13.9.2), or explicitly specialized (13.9.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (6.5.3.2) 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 is present if and only if the identifier I is preceded by inline. [Example:
namespace A::inline B::C {
    int i;
}
The above has the same effect as:

namespace A {
    inline namespace B {
        namespace C {
            int i;
        }
    }
}

—end example

9.8.1.1 Unnamed namespaces

An unnamed-namespace-definition behaves as if it were replaced by

```cpp
inline_opt namespace unique { /* empty body */ }
using namespace unique ;
namespace unique { namespace-body }
```

where 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:

```cpp
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]

9.8.1.2 Namespace member definitions

A declaration in a namespace N (excluding declarations in nested scopes) whose declarator-id is an unqualified-id (9.3.3), 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 (9.2.8.3), or that is an opaque-enum-declaration, declares (or redeclares) its unqualified-id or identifier as a member of N. [Note: An explicit instantiation (13.9.2) or explicit specialization (13.9.3) of a template does not introduce a name and thus may 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:

```cpp
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()
```
Members of a named namespace can also be defined outside that namespace by explicit qualification (6.5.3.2) 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:

```cpp
namespace Q {
    namespace V {
        void f();
    }
    void V::f() { /* ... */ } // OK
    void V::g() { /* ... */ } // error: g() is not yet a member of V
    namespace V {
        void g();
    }
}
namespace R {
    void Q::V::g() { /* ... */ } // error: R doesn't enclose Q
}
```

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 (6.5.1) or qualified lookup (6.5.3). [Note: 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 (6.5.2). 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: 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:

```cpp
// Assume f and g have not yet been declared.
void h(int);
template <class T> void f2(T);
namespace A {
    class Y {
        friend void f(X);    // A::f(X) is a friend
        friend void g();     // A::g is a friend
        friend void h(int);  // A::h is a friend
        friend void f2<>();   // ::f2<> 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
}
```

95) this implies that the name of the class or function is unqualified.
9.8.2 Namespace alias

A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

```
namespace-alias:
    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: When looking up a namespace-name in a namespace-alias-definition, only namespace names are considered, see 6.5.6. — end note]

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: 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]

9.8.3 Using namespace directive

A using-directive shall not appear in class scope, but may appear in namespace scope or in block scope. [Note: When looking up a namespace-name in a using-directive, only namespace names are considered, see 6.5.6. — end note] The optional attribute-specifier-seq appertains to the using-directive.

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 (6.5.1), the names appear as if they were declared in the nearest enclosing namespace which contains both the using-directive and the nominated namespace. [Note: In this context, “contains” means “contains directly or indirectly”. — end note]

A using-directive does not add any members to the declarative region in which it appears. [Example:

```
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
    }
```
```c
void f4() {
  i = 5;       // error: neither i is visible
}

—end example
```

For unqualified lookup (6.5.1), 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: For qualified lookup, see 6.5.3.2. — end note]

[Example:
```c
namespace M {
  int i;
}

namespace N {
  int i;
  using namespace M;
}

void f() {
  using namespace N;
  i = 7;       // error: both M::i and N::i are visible
}
```

For another example,
```c
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 (9.8.1) 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: 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,
```c
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: 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] 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. [Example:

```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++;       // OK
  D::d1++;       // OK
  d2++;         // OK: D::d2
  e++;          // OK: E::e
  f(l);         // 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 ...opt
  using-declarator-list , using-declarator ...opt

using-declarator:
  typename_opt nested-name-specifier unqualified-id
```

96) During name lookup in a class hierarchy, some ambiguities may 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.
Each using-declarator in a using-declaration\(^\text{97}\) 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 (6.5.3, 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: 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.

Every using-declaration is a declaration and a member-declaration and can therefore be used in a class definition. [Example:]

```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]

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:]

```cpp
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:

```cpp
template <typename... bases>
struct X : bases... {
    using bases::g...;
};
X<B, D> x; // OK: B::g and D::g introduced
```

—end example] [Example:

```cpp
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]

[Note: 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 (13.7.2). — end note] If a constructor or assignment operator brought from a base class into a derived class has the signature of a copy/move

---

\(^{97}\) A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using-declarator each.
constructor or assignment operator for the derived class (11.4.4.2, 11.4.5), 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:

```cpp
struct A {
    template <class T> void f(T);
    template <class T> struct X { }; // error
};
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:

```cpp
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 (6.5.3.2). [Example:

```cpp
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:

```cpp
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
};

— end example

[Note: 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 (9.3.3.6), default template arguments (13.2), and template specializations (13.7.5, 13.9.3) are considered. — end note]

Example:

namespace A {
    using A::f;
}

namespace A {
    void f(char);
}

namespace A {
    void f(int);
    void f(char);
}

void foo() {
    f('a');    // calls f(char), even though f(int) 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: 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 (13.7.5). — 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:

namespace A {
    int x;
}

namespace B {
    int i;
    struct g { };    // OK: hides struct g
    struct x { };    // OK: hides struct x
    void f(int);
    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]

If a function declaration in namespace scope or block scope has the same name and the same parameter-
type-list (9.3.3.5) 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: Two using-declarations may
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:

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'); // error: ambiguous: B::f(int) or C::f(int)?
    f(int); // error: f(int) conflicts with C::f(int) and B::f(int)
}
—end example] — end note]

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 (9.3.3.5), 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:

struct B {
    virtual void f(int);
    virtual void f(char);
    void g(int);
    void h(int);
};

struct D : B {
    using B::f; // OK: D::f(int) overrides B::f(int);
    using B::g;
    void g(char);
    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)
}

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]  

15 [Note: 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 (12.4.1). 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]

16 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 (6.5.3.1) or forming a set of overload candidates (12.4.1.3, 12.4.1.4, 12.4.1.7). [Note: 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 (11.10.3). A constructor of a derived class is sometimes preferred to a constructor of a base class if they would otherwise be ambiguous (12.4.3). —end note]

17 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.

18 [Note: 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:

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:

```c
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 (13.8.2), the name introduced by the *using-declaration* is treated as a *typedef-name* (9.2.3).

9.10 The *asm* declaration  

An *asm* declaration has the form

```c
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: Typically it is used to pass information through the implementation to an assembler. — end note]

9.11 Linkage specifications  

All function types, function names with external linkage, and variable names with external linkage have a language linkage. [Note: 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 may 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.

2 Linkage (6.6) between C++ and non-C++ code fragments can be achieved using a *linkage-specification*:

```c
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: Therefore, a linkage-specification with a *string-literal* that is unknown to the implementation requires a diagnostic. — end note] [Note: 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]

3 Every implementation shall provide for linkage to functions written in the C programming language, "C", and linkage to C++ functions, "C++". [Example:

```c
complex sqrt(complex); // C++ linkage by default
extern "C" {
    double sqrt(double); // C linkage
}
```
—end example]
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.

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:

```c
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

extern "C" typedef void FUNC();
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" {
  static void f4(); // the name of the function f4 has internal linkage (not C language linkage)
  // 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:

```c
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 (9.3.3.5) 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: Only one definition for an entity with a given name with C language linkage may 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:

```cpp
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 (9.2.1) 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:

```cpp
extern "C" double f();
static double f(); // error
extern "C" int i; // declaration
extern "C" {
  int i; // definition
}
extern "C" static void g(); // error
—end example]
```

[Note: Because the language linkage is part of a function type, when indirecting 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-namespaces :

attribute-namespaces:
  attribute-opt
  attribute-list , attribute-opt
  attribute ...
  attribute-list , attribute ...

attribute:
  attribute-token attribute-argument-clause-opt

attribute-token:
  identifier
  attribute-scoped-token

attribute-scoped-token:
  attribute-namespaces :: identifier

attribute-namespaces:
  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

2 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-namespaces specified in the attribute-using-prefix. [Note: This rule imposes no constraints on how an attribute-using-prefix affects the tokens in an attribute-argument-clause. — end note] [Example:

```
[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() {}
```

— end example]

3 [Note: For each individual attribute, the form of the balanced-token-seq will be specified. — end note]

4 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 (13.7.3). 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).

5 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 (11.9.3), that declaration shall be a definition. [Note: An attribute-specifier-seq cannot appertain to an explicit instantiation (13.9.2). — 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.

[Note: Each implementation should choose a distinctive name for the attribute-namespace in an attribute-scoped-token. — end note]

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: 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:

```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 (9.2.8.3) or class-head (Clause 11), respectively). An alignment-specifier with an ellipsis is a pack expansion (13.7.3).

1 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.

3 An alignment-specifier of the form `alignas(type-id)` has the same effect as `alignas(alignof(type-id))` (7.6.2.5).

4 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.

5 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. [Example:

```c
struct alignas(8) S {};
struct alignas(1) U {
    S s;
}; // error: U specifies an alignment that is less strict than if the alignas(1) were omitted.
```

— end example]

6 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:

```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;
—end example]

[Example: An aligned buffer with an alignment requirement of A and holding N elements of type T can be declared as:
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. —end example]

[Example:
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
—end example]

9.12.3 Carries dependency attribute [dcl.attr.depend]
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 (7.3.1) 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: The carries_dependency attribute does not change the meaning of the program, but may result in generation of more efficient code. —end note]

[Example:
/*/ 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;
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

The attribute-token `deprecated` can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. [Note: 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:

`attribute-list`

[Note: The string-literal in the `attribute-argument-clause` could 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: 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

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 (8.5.2). 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:

```c
void f(int n) {
    void g(), h(), i();
    switch (n) {
    case 1:
    case 2:
        g();
        [[fallthrough]];
```
case 3: // warning on fallthrough discouraged
    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 [dcl.attr.likelihood]

1 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: Excessive usage of either of these attributes is liable to result in performance degradation. —end note]

3 [Example:

```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.
A name or entity declared without the \texttt{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.

Recommended practice: For an entity marked \texttt{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 \texttt{maybe\_unused}, implementations should not emit such a warning unless all of its structured bindings are unused.

[Example:

```c
[[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 \texttt{b} is unused, whether or not \texttt{NDEBUG} is defined. —end example]

### 9.12.8 Nodiscard attribute [dcl\_attr\_nodiscard]

The attribute-token \texttt{nodiscard} may be applied to the \texttt{declarator-id} in a function declaration or to the declaration of a class or enumeration. It shall appear at most once in each \texttt{attribute-list}. An \texttt{attribute-argument-clause} may be present and, if present, shall have the form:

- ( \texttt{string-literal} )

A name or entity declared without the \texttt{nodiscard} attribute can later be redeclared with the attribute and vice-versa. [Note: Thus, an entity initially declared without the attribute can be marked as \texttt{nodiscard} by a subsequent redeclaration. However, after an entity is marked as \texttt{nodiscard}, later redeclarations do not remove the \texttt{nodiscard} from the entity. —end note] Redeclarations using different forms of the attribute (with or without the \texttt{attribute-argument-clause} or with different \texttt{attribute-argument-clauses}) are allowed.

A \texttt{nodiscard type} is a (possibly cv-qualified) class or enumeration type marked \texttt{nodiscard} in a reachable declaration. A \texttt{nodiscard call} is either

1. a function call expression (7.6.1.2) that calls a function declared \texttt{nodiscard} in a reachable declaration or whose return type is a nodiscard type, or
2. an explicit type conversion (7.6.1.3, 7.6.1.8, 7.6.3) that constructs an object through a constructor declared \texttt{nodiscard} in a reachable declaration, or that initializes an object of a nodiscard type.

Recommended practice: Appearance of a nodiscard call as a potentially-evaluated discarded-value expression (7.2) is discouraged unless explicitly cast to \texttt{void}. Implementations should issue a warning in such cases. [Note: This is typically because discarding the return value of a nodiscard call has surprising consequences. —end note] The \texttt{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:

```c
struct [[nodiscard]] my\_scopeguard { /* ... */ };  
struct my\_unique {  
    my\_unique() = default;  
    [[nodiscard]] my\_unique(int fd) { /* ... */ }  
    =my\_unique() noexcept { /* ... */ }  
    /* ... */  
};  
struct [[nodiscard]] error\_info { /* ... */ };  
error\_info enable\_missile\_safety\_mode();  
void launch\_missiles();  
void test\_missiles() {  
    my\_scopeguard();  
    (void)my\_scopeguard(),  
    launch\_missiles();  
    my\_unique(42);  
    my\_unique();  
    enable\_missile\_safety\_mode();  
    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 [dcl.attr.noreturn]
1 The attribute-token noreturn specifies that a function does not return. 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 in a function declaration. The first declaration of a function shall specify the noreturn attribute if any declaration of that function specifies the noreturn attribute. If a function is declared with the noreturn attribute in one translation unit and the same function is declared without the noreturn attribute in another translation unit, the program is ill-formed, no diagnostic required.

2 If a function f is called where f was previously declared with the noreturn attribute and f eventually returns, the behavior is undefined. [Note: The function may terminate by throwing an exception. — end note]

3 Recommended practice: Implementations should issue a warning if a function marked [[noreturn]] might return.

4 [Example:
    [[ noreturn ]] void f() {
        throw "error";     // OK
    }
    [[ 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 [dcl.attr.nouniqueaddr]
1 The attribute-token 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 attribute-list and no attribute-argument-clause shall be present. The attribute may appertain to a non-static data member other than a bit-field.

2 [Note: 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:
    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

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.

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.

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: 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]

[Example:

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 \texttt{A:Internals}, which does not contribute to the external interface of module \texttt{A}, and
(4.4) — a module implementation unit providing a definition of \texttt{bar} and \texttt{baz}, which cannot be imported because it does not have a partition name.

— end example

A \textit{module unit purview} is the sequence of tokens starting at the \texttt{module-declaration} and extending to the end of the translation unit. The \textit{purview} of a named module \texttt{M} is the set of module unit purviews of \texttt{M}'s module units.

The \textit{global module} is the collection of all \texttt{global-module-fragments} and all translation units that are not module units. Declarations appearing in such a context are said to be in the \textit{purview} of the global module. \[\textit{Note: The global module has no name, no module interface unit, and is not introduced by any \texttt{module-declaration}.} \]

A \textit{module} is either a named module or the global module. A declaration is \textit{attached} to a module as follows:

(7.1) — If the declaration
(7.1.1) — is a replaceable global allocation or deallocation function (17.6.2.1, 17.6.2.2), or
(7.1.2) — is a \texttt{namespace-definition} with external linkage, or
(7.1.3) — appears within a \texttt{linkage-specification},

it is attached to the global module.

(7.2) — Otherwise, the declaration is attached to the module in whose purview it appears.

A \texttt{module-declaration} that contains neither an \texttt{export-keyword} nor a \texttt{module-partition} implicitly imports the primary module interface unit of the module as if by a \texttt{module-import-declaration}. \[\textit{Example:}\]

Translation unit #1:

\begin{verbatim}
module B:Y;
int y();
\end{verbatim}

\texttt{// does not implicitly import B}

Translation unit #2:

\begin{verbatim}
export module B;
import :Y;
int n = y();
\end{verbatim}

\texttt{// OK, does not create interface dependency cycle}

Translation unit #3:

\begin{verbatim}
module B:X1;
int &a = n;
\end{verbatim}

\texttt{// does not implicitly import B}

\texttt{// error: n not visible here}

Translation unit #4:

\begin{verbatim}
module B:X2;
import B;
int &b = n;
\end{verbatim}

\texttt{// OK}

Translation unit #5:

\begin{verbatim}
module B;
int &c = n;
\end{verbatim}

\texttt{// implicitly imports B}

\texttt{// OK}

— end example

10.2 Export declaration

\begin{verbatim}
export-declaration:
  export declaration
  export { declaration-seq };
  export-keyword module-import-declaration
\end{verbatim}

An \texttt{export-declaration} shall appear only at namespace scope and only in the purview of a module interface unit. An \texttt{export-declaration} shall not appear directly or indirectly within an unnamed namespace or a \texttt{private-module-fragment}. An \texttt{export-declaration} has the declarative effects of its \texttt{declaration}, \texttt{declaration-seq} (if any), or \texttt{module-import-declaration}. An \texttt{export-declaration} does not establish a scope and its \texttt{declaration} or \texttt{declaration-seq} shall not contain an \texttt{export-declaration} or \texttt{module-import-declaration}.

A declaration is \textit{exported} if it is
(2.1) — a namespace-scope declaration declared within an export-declaration, or

(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.

3 An 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.

4 [Example:

Source file "a.h":

```c
export int x;
```

Translation unit #1:

```c
module;
#include "a.h" // error: declaration of x is not in the
// parview of a module interface unit
export module M;
export namespace {} // error: does not introduce any names
export namespace {
  int a1;
} 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 { } // OK
export using namespace N; // error: does not declare a name
```

—end example]

5 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:

Source file "b.h":

```c
int f();
```

Importable header "c.h":

```c
int g();
```

Translation unit #1:

```c
export module X;
export int h();
```

Translation unit #2:

```c
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: These constraints do not apply to type names introduced by typedef declarations and alias-declarations. [Example:

```c
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:

```c
export module M;
struct S { int n; };
typedef S S; // OK, does not redeclare an entity
export struct 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: 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:

Interface unit of M:

```c
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:

```c
module M;
struct A {
    int value;
};
```

Main program:

```c
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: Redeclaring a name in an export-declaration cannot change the linkage of the name (6.6).] [Example:

Interface unit of M:

```c
export module M;
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]

[Note: 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:

export module M;
export namespace N {
    int x;
    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 appertains to the module-import-declaration.

2 A module-import-declaration imports a set of translation units determined as described below. [Note: 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: All declarations within a header unit are implicitly exported (10.2), and are attached to the global module (10.1). An importable header is a member of an implementation-defined set of headers that includes all importable C++ library headers (16.5.1.2). 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: It is therefore possible that multiple copies exist of entities declared with internal linkage in an importable header. — end note]

[Note: 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. — end note]

6 A declaration of a name with internal linkage is permitted within a header unit despite all declarations being implicitly exported (10.2). [Note: 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.

7 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 may in turn lead to the importation of yet more translation units.

A module implementation unit shall not be exported. [Example:

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:

```
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:

Interface unit of \( M_1 \):
```
export module M1;
import M2;
```
Interface unit of \( M_2 \):
```
export module M2;
import M3;
```
Interface unit of \( M_3 \):
```
export module M3;
import M1;  // error: cyclic interface dependency \( M_3 \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \)
```
—end example]

10.4 Global module fragment  

```
global-module-fragment:
module-keyword ; declaration-seq
```

[module.global.frag]

1 [Note: Prior to phase 4 of translation, only preprocessing directives can appear in the declaration-seq (15.1). —end note]

2 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.

3 A declaration \( D \) is decl-reachable from a declaration \( S \) in the same translation unit if:

\[(3.1) \quad D \text{ does not declare a function or function template and } S \text{ contains an id-expression, namespace-name, type-name, template-name, or concept-name naming } D, \text{ or} \]

\[(3.2) \quad D \text{ declares a function or function template that is named by an expression (6.3) appearing in } S, \text{ or} \]

\[(3.3) \quad S \text{ contains an expression } E \text{ of the form}
\]
\[
\text{postfix-expression ( expression-list opt )}
\]
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

\[(3.4) \quad S \text{ contains an expression that takes the address of an overloaded function (12.5) whose set of overloads contains } D \text{ and for which the target type is dependent, or} \]

98) This is consistent with the rules for visibility of imported names (6.4.6).
there exists a declaration $M$ that is not a namespace-definition for which $M$ is decl-reachable from $S$ and either

- $D$ is decl-reachable from $M$, or
- $D$ 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
- $D$ declares a namespace $N$ and $M$ is a member of $N$, or
- one of $M$ and $D$ declares a class or class template $C$ and the other declares a member or friend of $C$, or
- one of $D$ and $M$ declares an enumeration $E$ and the other declares an enumerator of $E$, or
- $D$ declares a function or variable and $M$ is declared in $D$, or
- 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-declaration 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.

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: A discarded declaration is neither reachable nor visible to name lookup outside the module unit, nor in template instantiations whose points of instantiation (13.8.4.1) are outside the module unit, even when the instantiation context (10.6) includes the module unit. —end note]

[Example:

```c
const int size = 2;
int ary1[size]; // unspecified whether size is decl-reachable from ary1
cconstexpr 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;
cconstexpr int g(int);

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

—end example]

[Example:

Source file "foo.h":

```c
namespace N {
    struct X {};  // unspecified whether X is decl-reachable from foo
    int d();
    int e();
    inline int f(X, int = d()) { return e(); }
```
Module M interface:

```cpp
int g(X);
int h(X);
}

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]
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 {};
// definition not reachable from importers of A
X *factory() {
    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 (6.5.2) and which declarations are reachable (10.7) in the context of a particular declaration or template instantiation.

During the implicit definition of a defaulted function (11.4.3, 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.

During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (13.8.4.1), 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).

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.

During the instantiation of any other template specialization, the instantiation context comprises the point of instantiation of the template.

In any other case, the instantiation context at a point within the program comprises that point.

[Example:
Translation unit #1:
    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:
    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:
    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:
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

- the point at the end of translation unit #1,
- the point at the end of translation unit #2, and
- 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

- the point at the end of translation unit #1,
- the point at the end of translation unit #3, and
- 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

1 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: 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]

2 All translation units that are necessarily reachable are reachable. It is unspecified whether additional translation units on which the point within the program has an interface dependency are considered reachable, and under what circumstances. [Note: It is advisable to avoid depending on the reachability of any additional translation units in programs intending to be portable. — end note]

3 A declaration D is reachable if, for any point P in the instantiation context (10.6),

- D appears prior to P in the same translation unit, or
- 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.

[Note: Whether a declaration is exported has no bearing on whether it is reachable. — end note]

4 The accumulated properties of all reachable declarations of an entity within a context determine the behavior of the entity within that context. [Note: 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:

Translation unit #1:
    export module M:A;
    export struct B;

Translation unit #2:
    module M:B;

100) Implementations are therefore not required to prevent the semantic effects of additional translation units involved in the compilation from being observed.
struct B {
    operator int();
};

Translation unit #3:
module M:C;
import :A;
B b1;
   // error: no reachable definition of struct B

Translation unit #4:
export module M;
export import :A;
import :B;
B b2;
export void f(B b = B());

Translation unit #5:
module X;
import M;
B b3;
   // error: no reachable definition of struct B
void g() { f(); }  // error: no reachable definition of struct B
—end example] —-end note

[Note: An entity can have reachable declarations even if it is not visible to name lookup. — end note] [Example:

Translation unit #1:
export module A;
struct X();
export using Y = X;

Translation unit #2:
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-name or an elaborated-type-specifier (9.2.8.3) 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 (13.9.3) or a partial specialization (13.7.5). A class-specifier whose class-head omits the class-head-name defines an unnamed class. [Note: 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 (9.8.1) 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: 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:

```
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; } };  // OK: definition of nested class C.
  struct B final : C{}; // OK: definition of nested class B.
                      // not declaration of a bit-field member final
};
```

— end example]
[Note: Complete objects of class type have nonzero size. Base class subobjects and members declared with
the no_unique_address attribute (9.12.10) are not so constrained. — end note]

[Note: Class objects can be assigned (12.6.2.1, 11.4.5), passed as arguments to functions (9.4, 11.4.4.2), and
returned by functions (except objects of classes for which copying or moving has been restricted; see 9.5.3 and
11.9). Other plausible operators, such as equality comparison, can be defined by the user; see 12.6. — end note]

11.2 Properties of classes

A trivially copyable class is a class:

(1.1) — that has at least one eligible copy constructor, move constructor, copy assignment operator, or move
assignment operator (11.4.3, 11.4.4.2, 11.4.5),

(1.2) — where each eligible copy constructor, move constructor, copy assignment operator, and move assignment
operator is trivial, and

(1.3) — that has a trivial, non-deleted destructor (11.4.6).

A trivial class is a class that is trivially copyable and has one or more eligible default constructors (11.4.4.1),
all of which are trivial. [Note: In particular, a trivially copyable or trivial class does not have virtual functions
or virtual base classes. — end note]

A class \( \mathcal{S} \) is a standard-layout class if it:

(3.1) — has no non-static data members of type non-standard-layout class (or array of such types) or reference,

(3.2) — has no virtual functions (11.7.2) and no virtual base classes (11.7.1),

(3.3) — has the same access control (11.9) for all non-static data members,

(3.4) — has no non-standard-layout base classes,

(3.5) — has at most one base class subobject of any given type,

(3.6) — has all non-static data members and bit-fields in the class and its base classes first declared in the same
class, and

(3.7) — has no element of the set \( M(\mathcal{S}) \) of types as a base class, where for any type \( \mathcal{X} \), \( M(\mathcal{X}) \) is defined as
follows.\(^\text{101}\) [Note: \( M(\mathcal{X}) \) is the set of the types of all non-base-class subobjects that may be at a zero
offset in \( \mathcal{X} \). — end note]

3.7.1 — If \( \mathcal{X} \) is a non-union class type with no (possibly inherited (11.7)) non-static data members, the set
\( M(\mathcal{X}) \) is empty.

3.7.2 — If \( \mathcal{X} \) is a non-union class type with a non-static data member of type \( \mathcal{X}_0 \) that is either of zero size
or is the first non-static data member of \( \mathcal{X} \) (where said member may be an anonymous union), the set
\( M(\mathcal{X}) \) consists of \( \mathcal{X}_0 \) and the elements of \( M(\mathcal{X}_0) \).

3.7.3 — If \( \mathcal{X} \) is a union type, the set \( M(\mathcal{X}) \) is the union of all \( M(\mathcal{U}_i) \) and the set containing all \( \mathcal{U}_i \), where
each \( \mathcal{U}_i \) is the type of the \( i \)th non-static data member of \( \mathcal{X} \).

3.7.4 — If \( \mathcal{X} \) is an array type with element type \( \mathcal{X}_e \), the set \( M(\mathcal{X}) \) consists of \( \mathcal{X}_e \) and the elements of \( M(\mathcal{X}_e) \).

3.7.5 — If \( \mathcal{X} \) is a non-class, non-array type, the set \( M(\mathcal{X}) \) is empty.

[Example:

```
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
struct Q {}; // not a standard-layout class
struct S : Q { }; // not a standard-layout class
struct T : Q { }; struct U : S, T { }; // not a standard-layout class
```

— end example]

\(^{101}\) 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).
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.

[Note: Standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 11.4. — end note]

[Example:

```cpp
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]

7 [Note: Aggregates of class type are described in 9.4.1. — end note]

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

1 A class definition introduces a new type. [Example:

```cpp
struct X { int a; }
struct Y { int a; }
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```cpp
a1 = a2; // error: Y assigned to X
a1 = a3; // error: int assigned to X
```

are type mismatches, and that

```cpp
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,

```cpp
struct S { int a; }
struct S { int a; }; // error: double definition
```

is ill-formed because it defines S twice. — end example]

2 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 (6.5.4). [Example:
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()
}

—end example] 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:

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
}

—end example] [Note: Such declarations allow definition of classes that refer to each other. [Example:

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 11.9.3, operator functions in 12.6. — end example] — end note

3 [Note: An elaborated-type-specifier (9.2.8.3) can also be used as a type-specifier as part of a declaration. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. — end note] [Example:

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: The declaration of a class name takes effect immediately after the identifier is seen in the class definition or elaborated-type-specifier. For example,

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
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 (11.5.1) and direct members thereof. Members of a class are data members, member functions (11.4.1), nested types, enumerators, and member templates (13.7.2) and specializations thereof. [Note: 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 (11.9.3),
(2.2) — a static_assert-declaration,
(2.3) — a using-declaration (9.9), or
(2.4) — an empty-declaration.

For any other member-declaration, each declared entity that is not an unnamed bit-field (11.4.9) 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, 11.4.10) and enumerations (9.7.1) declared in the class and arbitrary types declared as members by use of a typedef declaration (9.2.3) 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 11.4.8) (a static data member (11.4.8.2) or static member function (11.4.8.1), 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 (11.4.2), respectively). [Note: A non-static data member of non-reference type is a member subobject of a class object (6.7.2). —end note]
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: 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 (9.3.3.6),
(6.3) — noexcept-specifier (14.5), or
(6.4) — default member initializer

within the member-specification of the class. [Note: 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 = 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:

```cpp
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:

```cpp
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 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]
```

10 A brace-or-equal-initializer shall appear only in the declaration of a data member. (For static data members, see 11.4.8.2; for non-static data members, see 11.10.2 and 9.4.1). 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.

11 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.

12 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 (11.9.3). A pure-specifier shall be used only in the declaration of a virtual function (11.7.2) that is not a friend declaration.

13 The optional attribute-specifier-seq in a member-declaration appertains to each of the entities declared by the member-declarations; 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 (11.7.2).

The type of a non-static data member shall not be an incomplete type (6.8), an abstract class type (11.7.3), or a (possibly multi-dimensional) array thereof. [Note: 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: See 7.5.4 for restrictions on the use of non-static data members and non-static member functions. — end note]

[Note: 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: A simple example of a class definition is

```cpp
class 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

```cpp
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: 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. 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 (11.7.2) and virtual base classes (11.7.1). — end note]

If `T` is the name of a class, then each of the following shall have a name different from `T`:

- every static data member of class `T`;
- every member function of class `T` [Note: This restriction does not apply to constructors, which do not have names (11.4.4) — end note];
- every member of class `T` that is itself a type;
- every member template of class `T`;
- every enumerator of every member of class `T` that is an unscoped enumerated type; and
- every member of every anonymous union that is a member of class `T`.

In addition, if class `T` has a user-declared constructor (11.4.4), 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:

```cpp
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 structs (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 \( T_1 \), it is permitted to read a non-static data member \( m \) of another union member of struct type \( T_2 \) provided \( m \) is part of the common initial sequence of \( T_1 \) and \( T_2 \); the behavior is as if the corresponding member of \( T_1 \) were nominated. [Example:

```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: Reading a volatile object through a glvalue of non-volatile type has undefined behavior (9.2.8.1). —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: 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: The object and its first subobject are pointer-interconvertible (6.8.2, 7.6.1.8). —end note]

### 11.4.1 Member functions [class.mfct]

1 A member function may be defined (9.5) in its class definition, in which case it is an inline (9.2.7) 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: A member function is also inline if it is declared `inline`, `constexpr`, or `consteval`. —end note]

2 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: There can be at most one definition of a non-inline member function in a program. There may be more than one inline member function definition in a program. See 6.3 and 9.2.7. —end note] [Note: Member functions of a class have the linkage of the name of the class. See 6.6. —end note]

3 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: A name used in a member function definition (that is, in the parameter-declaration-clause including the default arguments (9.3.3.6) or in the member function body) is looked up as described in 6.5. —end note] [Example:

```c
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: 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]

5 Previously declared member functions may be mentioned in friend declarations.

6 Member functions of a local class shall be defined inline in their class definition, if they are defined at all.
Note: 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 9.3.3.5. 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;
fv S::* pmfv2 = &S::memfunc2;
fvc S::* pmfv3 = &S::memfunc3;

Also see 13.4. — end note

11.4.2 Non-static member functions

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 (7.6.1.4, 12.4.1.1). A non-static member function may also be called directly using the function call syntax (7.6.1.2, 12.4.1.1) 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 (7.6.1.4) and not used to form a pointer to member (7.6.2.1) 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 (7.6.1.4) using (\*this) (11.4.2.1) as the postfix-expression to the left of the . operator. [Note: 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 (13.8.2.1).

[Example:

```c
struct tnode {
  char tword[20];
  int count;
  tnode* left;
  tnode* right;
  void set(const char*, tnode* l, tnode* r);
};

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.102 — end example]

4 A non-static member function may be declared const, volatile, or const volatile. These cv-qualifiers affect the type of the this pointer (11.4.2.1). They also affect the function type (9.3.3.5) of the member

102 See, for example, <cstring> (21.5.3).
function; a member function declared \texttt{const} is a \textit{const member function}, a member function declared \texttt{volatile} is a \textit{volatile member function} and a member function declared \texttt{const volatile} is a \textit{const volatile member function}. \[\text{Example:}\]

```c
struct X {
    void g() const;  
    void h() const volatile;
}
```

\texttt{X::g} is a \textit{const member function} and \texttt{X::h} is a \textit{const volatile member function}. \[\text{— end example}\]

A non-static member function may be declared with a \texttt{ref-qualifier} (9.3.3.5); see 12.4.1.

A non-static member function may be declared virtual (11.7.2) or pure virtual (11.7.3).

### 11.4.2.1 The this pointer

In the body of a non-static (11.4.1) member function, the keyword \texttt{this} is a prvalue whose value is a pointer to the object for which the function is called. The type of \texttt{this} in a member function whose type has a \texttt{cv-qualifier-seq cv} and whose class is \texttt{X} is “pointer to \texttt{cv X}”. \[\text{Note:} \texttt{this} is a pointer to \texttt{const} when accessed through a const access path. \[\text{— end note}\]

```c
struct s {
    int a;
    int f() const;
    int g() { return a++; }  // error
    int h() const { return a++; }  // error
};
```

\texttt{int s::f()} is a \textit{non-const member function}, that is, \texttt{h()} is less-qualified than the object expression \texttt{s}. \[\text{— end note}\]

A member function whose type has a \texttt{cv-qualifier-seq cv} can be called on an object expression (7.6.1.4) of type \texttt{cv2 T} only if \texttt{cv1} is the same as or more \texttt{cv}-qualified than \texttt{cv2} (6.8.3). \[\text{Example:}\]

```c
void k(s& x, const s& y) {
    x.f();
    x.g();
    y.f();
    y.g();
    // error
}
```

\texttt{y.g()} is ill-formed because \texttt{y} is \texttt{const} and \texttt{s::g()} is a non-const member function, that is, \texttt{s::g()} is less-qualified than the object expression \texttt{y}. \[\text{— end example}\]

\[\text{Note:} \texttt{this} is a pointer to \texttt{const} when accessed through a const access path. \[\text{— end note}\]

### 11.4.3 Special member functions

Default constructors (11.4.4.1), copy constructors, move constructors (11.4.4.2), copy assignment operators, move assignment operators (11.4.5), and prospective destructors (11.4.6) are \textit{special member functions}. \[\text{Note:} \texttt{this} is a pointer to \texttt{const} when accessed through a const access path. \[\text{— end note}\]

Programs may explicitly refer to implicitly-declared special member functions. \[\text{Example:} \texttt{A program may explicitly call or form a pointer to member to an implicitly-declared special member function.}\]

```c
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

[Note: 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: Declaring a constructor protected
ensures that only derived classes and friends can create objects using it. — end example]

Two special member functions are of the same kind if:

— they are both default constructors,
— they are both copy or move constructors with the same first parameter type, or
— they are both copy or move assignment operators with the same first parameter type and the same
cv-qualifiers and ref-qualifier, if any.

An eligible special member function is a special member function for which:

— the function is not deleted,
— the associated constraints (13.5), if any, are satisfied, and
— no special member function of the same kind is more constrained (13.5.4).

For a class, its non-static data members, its non-virtual direct base classes, and, if the class is not abstract
(11.7.3), its virtual base classes are called its potentially constructed subobjects.

A defaulted special member function is constexpr-compatible if the corresponding implicitly-declared special
member function would be a constexpr function.

11.4.4 Constructors

A constructor is introduced by a declaration whose declarator is a function declarator (9.3.3.5) of the form

ptr-declarator ( parameter-declaration-clause ) noexcept-specifier_opt attribute-specifier-seq_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:

— in a member-declaration that belongs to the member-specification of a class or class template but is not a
friend declaration (11.9.3), the id-expression is the injected-class-name (11.1) of the immediately-enclosing
entity or
— in a declaration at namespace scope or in a friend declaration, the id-expression is a
qualified-id that
names a constructor (6.5.3.1).

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:

struct S {
    S();                               // declares the constructor
};
S::S() { }                          // defines the constructor
— end example

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 (7.6.1.3)
will cause a constructor to be called to initialize an object. [Note: The syntax looks like an explicit call of
the constructor. — end note] [Example:

complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );

§ 11.4.4
An object created in this way is unnamed. \[\text{Note: 6.7.7 describes the lifetime of temporary objects.} \quad \text{— end note}\]

A constructor can be invoked for a \texttt{const}, \texttt{volatile} or \texttt{const volatile} object. \texttt{const} and \texttt{volatile} semantics (9.2.8.1) are not applied on an object under construction. They come into effect when the constructor for the most derived object (6.7.2) ends.

A \texttt{return} statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

A constructor shall not be a coroutine.

### 11.4.4.1 Default constructors

A \texttt{default constructor} for a class \texttt{X} is a constructor of class \texttt{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 \texttt{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.

A defaulted default constructor for class \texttt{X} is defined as deleted if:

1. \texttt{X} is a union that has a variant member with a non-trivial default constructor and no variant member of \texttt{X} has a default member initializer,
2. \texttt{X} is a non-union class that has a variant member \texttt{M} with a non-trivial default constructor and no variant member of the anonymous union containing \texttt{M} has a default member initializer,
3. any non-static data member with no default member initializer (11.4) is of reference type,
4. any non-variant non-static data member of const-qualified type (or array thereof) with no \texttt{brace-or-equal-initializer} is not const-default-constructible (9.4),
5. \texttt{X} is a union and all of its variant members are of const-qualified type (or array thereof),
6. \texttt{X} is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof),
7. any potentially constructed subobject, except for a non-static data member with a \texttt{brace-or-equal-initializer}, has class type \texttt{M} (or array thereof) and either \texttt{M} has no default constructor or overload resolution (12.4) as applied to find \texttt{M}'s corresponding constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or
8. any potentially constructed subobject has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.

A default constructor is \texttt{trivial} if it is not user-provided and if:

1. its class has no virtual functions (11.7.2) and no virtual base classes (11.7.1), and
2. no non-static data member of its class has a default member initializer (11.4), and
3. all the direct base classes of its class have trivial default constructors, and
4. 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 \texttt{non-trivial}.

A default constructor that is defaulted and not defined as deleted is \textit{implicitly defined} when it is \texttt{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 \texttt{ctor-initializer} (11.10.2) and an empty \texttt{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 \texttt{constexpr} constructor (9.2.5), the implicitly-defined default constructor is \texttt{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. \[\text{Note: An implicitly-declared default constructor is \texttt{constexpr} if it satisfies the requirements of a \texttt{constexpr} constructor (9.2.5).} \quad \text{— end note}\]
constructor has an exception specification (14.5). An explicitly-defaulted definition might have an implicit exception specification, see 9.5. —end note]

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (6.7.5.1, 6.7.5.2, 6.7.5.3) defined without an initializer (9.4), are called to create class objects of dynamic storage duration (6.7.5.4) created by a new-expression in which the new-initializer is omitted (7.6.2.7), or are called when the explicit type conversion syntax (7.6.1.3) 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).

[Note: 11.10.2 describes the order in which constructors for base classes and non-static data members are called and describes how arguments can be specified for the calls to these constructors. —end note]

11.4.4.2 Copy/move constructors [class.copy.ctor]

1 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 (9.3.3.6). [Example: X::X(const X&) and X::X(X&, int=1) are copy constructors.

```cpp
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);
—end example]
```

2 A non-template constructor for class X is a move 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 (9.3.3.6). [Example: 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]
```

3 [Note: All forms of copy/move constructor may be declared for a class. [Example:

```cpp
struct X {
    X(const X&);
    X(X&); // OK
    X(X&&); // OK, but possibly not sensible
};
—end example] —end note]
```

4 [Note: If a class X only has a copy constructor with a parameter of type X&, an initializer of type const X or volatile X cannot initialize an object of type cv X. [Example:

```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]
```

5 A declaration of a constructor for a class X is ill-formed if its first parameter is of type 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:

```cpp
struct S {
    template<typename T> S(T);
}
```
© ISO/IEC  N4861

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

6 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.8).

7 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&. Otherwise, the implicitly-declared copy constructor will have the form

X::X(X&)

8 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: When the move constructor is not implicitly declared or explicitly supplied, expressions that otherwise would have invoked the move constructor may instead invoke a copy constructor. — end note]

9 The implicitly-declared move constructor for class X will have the form

X::X(X&&)

10 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: 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]

11 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 (11.7.2) and no virtual base classes (11.7.1), 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.

103) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.5.6.

§ 11.4.4.2 260
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: 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 (9.2.5), 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: 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: Default member initializers of non-static data members are ignored. See also the example in 11.10.2. — end note] The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 11.10.2). Let \(x\) be either the parameter of the constructor or, for the move constructor, an xvalue 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 rvalue reference type \(T&&\), it is direct-initialized with 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 11.10.2).

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.

### 11.4.5 Copy/move assignment operator

A user-declared copy assignment operator \(X::\text{operator}=(\cdot)\) is a non-static non-template member function of class \(X\) with exactly one parameter of type \(X, X&, \text{const } X&, \text{volatile } X&, \text{or const volatile } X&.\) [Note: An overloaded assignment operator must be declared to have only one parameter; see 12.6.2.1. — end note] [Note: More than one form of copy assignment operator may be declared for a class. — end note] [Note: 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:

```c
struct X {
    X();
    X& operator=(X&);  // error: X::operator=(X&) cannot assign cx into x
};

void f() {
    const X cx;
    X x;
    x = cx;   // error: X::operator=(X&) cannot assign cx into x
}
```

— end example] — end note]

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.8). The implicitly-declared copy assignment operator for a class \(X\) will have the form

\[
X& \ X::\text{operator}=(\text{const } X&)
\]

if

1. each direct base class \(B\) of \(X\) has a copy assignment operator whose parameter is of type \(\text{const } B&\), \(\text{const volatile } B&\), or \(B\), and

---

104) 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.
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 \( \text{const } M& \), \( \text{const volatile } M& \), or
\( M \). Otherwise, the implicitly-declared copy assignment operator will have the form

\[ X& \ X::\text{operator}= (X&) \]

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: An overloaded assignment operator must be declared to have only one parameter; see 12.6.2.1. — end note]

[Note: More than one form of move assignment operator may 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

1. \( X \) does not have a user-declared copy constructor,
2. \( X \) does not have a user-declared move constructor,
3. \( X \) does not have a user-declared copy assignment operator, and
4. \( X \) does not have a user-declared destructor.

[Example: The class definition

\[
\text{struct } S \{ \\
\quad \text{int } a; \\
\quad S& \ \text{operator}= (\text{const } S&) = \text{default}; \\
\};
\]

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.

\[
\text{struct } S \{ \\
\quad \text{int } a; \\
\quad S& \ \text{operator}= (\text{const } S&) = \text{default}; \\
\quad S& \ \text{operator}= (S&&) = \text{default}; \\
\};
\]

—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:

1. a variant member with a non-trivial corresponding assignment operator and \( X \) is a union-like class, or
2. a non-static data member of \( \text{const} \) non-class type (or array thereof), or
3. a non-static data member of reference type, or
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: 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 (12.6.2.1). A 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

105) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a \text{volatile}
lvalue; see C.5.6.
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 (11.7.2) and no virtual base classes (11.7.1), 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;

otherwise the copy/move assignment operator is non-trivial.

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:

- (10.1) \(X\) is a literal type, and
- (10.2) the assignment operator selected to copy/move each direct base class subobject is a constexpr function, and
- (10.3) 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 a constexpr function.

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: An implicitly-declared copy/move assignment operator has an implied exception specification (14.5). —end note]

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:

- (12.1) if the subobject is of class type, as if by a call to \(\text{operator=}\) with the subobject as the object expression and the corresponding subobject of \(x\) as a single function argument (as if by explicit qualification; that is, ignoring any possible virtual overriding functions in more derived classes);
- (12.2) if the subobject is an array, each element is assigned, in the manner appropriate to the element type;
- (12.3) if the subobject is of scalar type, the built-in assignment operator is used.

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy/move assignment operator. [Example:

```c
struct V {}; struct A : virtual V {}; struct B : virtual V {}; struct C : B, A {};  
```

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]

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.

### 11.4.6 Destructors [class.dtor]

A prospective destructor is introduced by a declaration whose declarator is a function declarator (9.3.3.5) of the form

\[
\text{ptr-declarator} \ (\text{parameter-declaration-clause}) \ \text{noexcept-specifier_opt} \ \text{attribute-specifier-seq_opt}
\]

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:
in a member-declaration that belongs to the member-specification of a class or class template but is not a friend declaration (11.9.3), the id-expression is ~class-name and the class-name is the injected-class-name (1.1) of the immediately-enclosing entity or

— in a declaration at namespace scope or in a friend declaration, the id-expression is nested-name-specifier ~class-name and the class-name names the same class as the nested-name-specifier.

A prospective destructor shall take no arguments (9.3.3.5). 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

~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 (9.2.8.1) 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:

— X is a union-like class that has a variant member with a non-trivial destructor,

— 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,

— 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:

— the destructor is not virtual,

— all of the direct base classes of its class have trivial destructors, and

— 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 (9.2.5).

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 deleted 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 (11.7.2) or pure virtual (11.7.3). 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.

[Note: Some language constructs have special semantics when used during destruction; see 11.10.4. — end note]

After executing the body of the destructor and destroying any objects with automatic storage duration allocated within the body, a destructor for class X calls the destructors for X’s direct non-variant non-static data members, the destructors for X’s non-virtual direct base classes and, if X is the most derived class (11.10.2), its destructor calls the destructors for X’s virtual base classes. All destructors are called as if they were referenced with a qualified name, that is, ignoring any possible virtual overriding destructors in more derived classes. Bases and members are destroyed in the reverse order of the completion of their constructor (see 11.10.2).
return statement (8.7.3) in a destructor might not directly return to the caller; before transferring control to the caller, the destructors for the members and bases are called. Destructors for elements of an array are called in reverse order of their construction (see 11.10).

15 A destructor is invoked implicitly

(15.1) — for a constructed object with static storage duration (6.7.5.1) at program termination (6.9.3.4),
(15.2) — for a constructed object with thread storage duration (6.7.5.2) at thread exit,
(15.3) — for a constructed object with automatic storage duration (6.7.5.3) when the block in which an object is created exits (8.8),
(15.4) — for a constructed temporary object when its lifetime ends (7.3.4, 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 (7.6.2.8) for a constructed object allocated by a new-expression (7.6.2.7); the context of the invocation is the delete-expression. [Note: 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 7.6.2.7, 8.7.3, 9.4.1, 11.10.2, 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.

16 At the point of definition of a virtual destructor (including an implicit definition (11.4.6)), 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 7.6.2.8). If the lookup fails or if the deallocation function has a deleted definition (9.5), the program is ill-formed. [Note: This assures that a deallocation function corresponding to the dynamic type of an object is available for the delete-expression (11.12). — end note]

17 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 (11.4.1); 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: Invoking delete on a null pointer does not call the destructor; see 7.6.2.8. — end note] [Example:

```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 B’s destructor
    B_ptr->B_alias::~B();     // 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: An explicit destructor call must always be written using a member access operator (7.6.1.4) or a qualified-id (7.5.4.2); in particular, the unary-expression ~X() in a member function is not an explicit destructor call (7.6.2.1). — end note]

[Note: 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);
}
```

§ 11.4.6 265
Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (6.7.3). [Example: 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]

[Note: The notation for explicit call of a destructor can be used for any scalar type name (7.5.4.3). Allowing this makes it possible to write code without having to know if a destructor exists for a given type. For example:

```c++
typedef int I;
I* p;
p->I::~I();
— end note]
```

A destructor shall not be a coroutine.

11.4.7 Conversions

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 (7.6.1.3, 7.6.3, 7.6.1.8).

User-defined conversions are applied only where they are unambiguous (11.8, 11.4.7.2). Conversions obey the access control rules (11.9). Access control is applied after ambiguity resolution (6.5).

[Note: See 12.4 for a discussion of the use of conversions in function calls as well as examples below. — end note]

At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value. [Example:

```c++
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]
```

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 (12.4.3) selects the best conversion function to perform the conversion. [Example:

```c++
struct X {
    operator int();
};
```
11.4.7.1 Conversion by constructor

A constructor that is not explicit (9.2.2) 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:

```cpp
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]

11.4.7.2 Conversion functions

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 (9.3.3.5) 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.106 [Example:

```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 (9.2.2), 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:

```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: This prevents ambiguities between the declarator operator * and its expression counterparts. [Example:

```c
&ac.operator int*+1; // syntax error:
// parsed as: &ac.operator int *+1
// not as: &ac.operator int)*1
```

The * is the pointer declarator and not the multiplication operator. — end example] This rule also prevents ambiguities for attributes. [Example:

```c
operator int [[noreturn]] (); // error: noreturn attribute applied to a type
```

— end example] — end note]

4 Conversion functions are inherited.

5 Conversion functions can be virtual.

106) These conversions are considered as standard conversions for the purposes of overload resolution (12.4.3.1, 12.4.3.1.4) and therefore initialization (9.4) and explicit casts (7.6.1.8). A conversion to void does not invoke any conversion function (7.6.1.8). 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.
A conversion function template shall not have a deduced return type (9.2.8.5). [Example:

```cpp
struct S {
    operator auto() const { return 10; } // OK
    template<class T>
    operator auto() const { return 1.2; } // error: conversion function template
};
```

—end example]

### 11.4.8 Static members [class.static]

A static member `s` of class `X` may be referred to using the `qualified-id` expression `X::s`; it is not necessary to use the class member access syntax (7.6.1.4) to refer to a static member. A static member may be referred to using the class member access syntax, in which case the object expression is evaluated. [Example:

```cpp
struct process {
    static void reschedule();
};
process& g();

void f() {
    process::reschedule(); // OK: no object necessary
    g().reschedule(); // g() is called
}
```

—end example]

A static member may be referred to directly in the scope of its class or in the scope of a class derived (11.7) from its class; in this case, the static member is referred to as if a `qualified-id` expression was used, with the `nested-name-specifier` of the `qualified-id` naming the class scope from which the static member is referenced. [Example:

```cpp
int g();
struct X {
    static int g();
};
struct Y : X {
    static int i;
};
int Y::i = g(); // equivalent to Y::g();
```

—end example]

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: It cannot be specified in member declarations that appear in namespace scope. —end note]

#### 11.4.8.1 Static member functions [class.static.mfct]

[Note: The rules described in 11.4.1 apply to static member functions. —end note]

[Note: A static member function does not have a `this` pointer (11.4.2.1). —end note] A static member function shall not be `virtual`. There shall not be a static and a non-static member function with the same name and the same parameter types (12.2). A static member function shall not be declared `const`, `volatile`, or `const volatile`.

#### 11.4.8.2 Static data members [class.static.data]

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` (9.2.1). 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 (11.4.10) 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

§ 11.4.8.2 269
at namespace scope, the name of the static data member shall be qualified by its class name using the :: operator. The \textit{initializer} expression in the definition of a static data member is in the scope of its class (6.4.7).

\textbf{Example:}

```c
class process {
    static process* run_chain;
    static process* running;
};
```

```c
process* process::run_chain = get_main();
process* process::run_chain = running;
```

The static data member \texttt{run\_chain} of class \texttt{process} is defined in global scope; the notation \texttt{process::run\_chain} specifies that the member \texttt{run\_chain} is a member of class \texttt{process} and in the scope of class \texttt{process}. In the static data member definition, the \textit{initializer} expression refers to the static data member \texttt{running} of class \texttt{process}. \textit{— end example}\]

\[\text{Note:} \text{ Once the static data member has been defined, it exists even if no objects of its class have been created.} \text{ \textit{Example:} In the example above,} \texttt{run\_chain} \text{ and} \texttt{running} \text{ exist even if no objects of class} \texttt{process} \text{ are created by the program.} \textit{— end example}\]

4 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 \texttt{constexpr} specifier, it may be redeclared in namespace scope with no initializer (this usage is deprecated; see D.6). Declarations of other static data members shall not specify a brace-or-equal-initializer.

5 \[\text{Note:} \text{ There is exactly one definition of a static data member that is odr-used (6.3) in a valid program.} \textit{— end note}\]

6 \[\text{Note:} \text{ Static data members of a class in namespace scope have the linkage of the name of the class (6.6).} \textit{— end note}\]

7 Static data members are initialized and destroyed exactly like non-local variables (6.9.3.2, 6.9.3.3, 6.9.3.4).

\section*{11.4.9 Bit-fields \texttt{[class.bit]}}

A member-declarator of the form

\texttt{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. \[\text{Note:} \text{ Bit-fields straddle allocation units on some machines and not on others.} \text{ Bit-fields are assigned right-to-left on some machines, left-to-right on others.} \textit{— end note}\]

2 A declaration for a bit-field that omits the \texttt{identifier} declares an \textit{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. \[\text{Note:} \text{ An unnamed bit-field is useful for padding to conform to externally-imposed layouts.} \textit{— end note}\]

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.

3 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 (9.4.3). \[\text{Note:} \text{ If the initializer for a reference of type} \texttt{const Tk} \text{ is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly.} \texttt{See} 9.4.3. \textit{— end note}\]

4 If a value of integral type (other than \texttt{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 \texttt{true} or \texttt{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:

```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]

11.4.10 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: See 7.5.4 for restrictions on the use of non-static data members and non-static member functions. —end note]

[Example:

```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  
    }  
  }  

  inner* p = 0;  // error: inner not in scope  
};
```
—end example]

2 Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [Example:

```c
struct enclose {  
  struct inner {  
    static int x;  
    void f(int i);  
  };  
};  

int enclose::inner::x = 1;  

void enclose::inner::f(int i) { /* ... */ }
```
—end example]
3 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:

```c++
class E {
   class I1; // forward declaration of nested class
   class I2;
   class I1 {}; // definition of nested class
   class E::I2 {}; // definition of nested class
   
} - end example
```

4 Like a member function, a friend function (11.9.3) 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 (11.4.8), but it has no special access rights to members of an enclosing class.

### 11.4.11 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:

```c++
struct X {
   typedef int I;
   class Y { /* ... */ }; // forward declaration of nested class
   I a;
};

I b; // error
Y c; // error
X::Y d; // OK
X::I e; // OK

- end example
```

### 11.5 Unions

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: 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 structs, it is permitted to inspect the common initial sequence of any of the standard-layout struct members; see 11.4. — end note]

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: A union object and its non-static data members are pointer-interconvertible (6.8.2, 7.6.1.8). As a consequence, all non-static data members of a union object have the same address. — end note]

A union can have member functions (including constructors and destructors), but it shall not have virtual (11.7.2) 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: Absent default member initializers (11.4), if any non-static data member of a union has a non-trivial default constructor (11.4.4.1), copy constructor, move constructor (11.4.4.2), copy assignment operator, move assignment operator (11.4.5), or destructor (11.4.6), 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]

[Example: Consider the following union:

```c++
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]

When the left operand of an assignment operator involves a member access expression (7.6.1.4) 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:

1. If `E` is of the form `A.B`, \( S(E) \) contains the elements of \( S(A) \), and also contains `A.B` if `B` 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.
2. If `E` is of the form `A[B]` and is interpreted as a built-in array subscripting operator, \( S(E) \) is \( S(A) \) if `A` is of array type, \( S(B) \) if `B` is of array type, and empty otherwise.
3. Otherwise, \( S(E) \) 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 (11.4.5), 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: This ends the lifetime of the previously-active member of the union, if any (6.7.3). — end note]

Example:

```cpp
union A { int x; int y[4]; }
struct B { A a; }
union C { B b; int k; }
int f() {
    union A 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() {
    union Y y = { { 1, 2 } };     // OK, y.x is active union member (11.4)
    int n = y.x.a;
    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,
    // S(y.x.b) is empty because X's default constructor is deleted,
    // so union member y.x's lifetime does not implicitly start
}
```

— end example]

[Note: In general, one must use explicit destructor calls and placement `new-expression` to change the active member of a union. — end note] [Example: Consider an object `u` of a 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:

```cpp
u.m.-M();
new (ku.n) N;
```
— end example]

11.5.1 Anonymous unions

A union of the form

```cpp
union { member-specification } ;
```

is called an anonymous union; it defines an unnamed type and an unnamed object of that type called an anonymous union object. Each member-declaration in the member-specification of an anonymous union shall either define a non-static data member or be a 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

§ 11.5.1
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:

```c
void f() {
    union { int a; const char* p; };  
    a = 1;  
    p = "Jennifer";
}
```

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 `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. An anonymous union shall not have private or protected members (11.9). An anonymous union shall not have member functions.

A union for which objects, pointers, or references are declared is not an anonymous union. [Example:

```c
void f() {
    union { int aa; char* p; } obj, *ptr = &obj;  
    aa = 1; // error  
    ptr->aa = 1; // OK
}
```

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] [Note: Initialization of unions with no user-declared constructors is described in 9.4.1. —end note]

A 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 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:

```c
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 scope, and has the same access to names outside the function as does the enclosing function. [Note: A declaration in a local class cannot odr-use (6.3) a local entity from an enclosing scope. —end note] [Example:

```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: A local class cannot have static data members (11.4.8.2). —end note]

11.7 Derived classes

A list of base classes can be specified in a class definition using the notation:

```
base-clause:
   : base-specifier-list

base-specifier-list:
   base-specifier ... opt
   base-specifier-list , base-specifier ... opt

base-specifier:
   attribute-specifier-seq opt class-or-decltype
   attribute-specifier-seq opt virtual access-specifier_opt class-or-decltype
   attribute-specifier-seq opt access-specifier virtual_opt class-or-decltype

class-or-decltype:
   nested-name-specifier_opt type-name
   nested-name-specifier template simple-template-id
decltype-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: 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: The scope resolution operator :: (7.5.4.2) 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 11.9.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (7.3.11). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (9.4.3). —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:

§ 11.7
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 (13.7.3).

5 The order in which the base class subobjects are allocated in the most derived object (6.7.2) is unspecified. [Note: 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]

![Directed acyclic graph](fig:class.dag)

6 [Note: Initialization of objects representing base classes can be specified in constructors; see 11.10.2. — end note]

7 [Note: A base class subobject might have a layout (6.7.5) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (11.10.4) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause 11); however, two subobjects that have the same class type and that belong to the same most derived object must not be allocated at the same address (7.6.10). — end note]

11.7.1 Multiple base classes [class.mi]

1 A class can be derived from any number of base classes. [Note: The use of more than one direct base class is often called multiple inheritance. — end note] [Example:

```cpp
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
— end example]
```

2 [Note: The order of derivation is not significant except as specified by the semantics of initialization by constructor (11.10.2), cleanup (11.4.6), and storage layout (11.4, 11.9.1). — end note]

3 A class shall not be specified as a direct base class of a derived class more than once. [Note: 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:

```cpp
class X { /* ... */ };
class Y : public X, public X { /* ... */ }; // error
```

— 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: 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).

[End note]

[Note: 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. — end note]

[Note: 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]

11.7.2 Virtual functions [class.virtual]

1 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).107 [Note: Virtual functions support dynamic binding and object-oriented programming. — end note] A class that declares or inherits a virtual function is called a polymorphic class.108

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 (9.3.3.5), 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:

```c
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:

```c
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
```

107) The use of the virtual specifier in the declaration of an overriding function is valid but redundant (has empty semantics).

108) 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.

109) 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.
— end example]

3 [Note: 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:

```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]

5 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:

```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]

6 A virtual function shall not have a trailing requires-clause (9.3). [Example:

```cpp
struct A {
    virtual void f() requires true; // error: virtual function cannot be constrained (13.5.2)
};
```

— end example]

7 Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 11.4.6 and 11.12.

8 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.

110) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.
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 (7.6.1.2). [Example:

```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
    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]

[Note: 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) (7.6.1.2). —end note]

[Note: The \texttt{virtual} specifier implies membership, so a virtual function cannot be a non-member (9.2.2) 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 (11.9.3) in another class. —end note]

A virtual function declared in a class shall be defined, or declared pure (11.7.3) in that class, or both; no diagnostic is required (6.3).

[Example: 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* bip = &d;
    A* ap = bip;
    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: The following example shows a function that does not have a unique final overrider:

    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: 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* vblap = new Da;
        vblap->f(); // calls VB2::f
    }

    — end example]

16 Explicit qualification with the scope operator (7.5.4.2) suppresses the virtual call mechanism. [Example:

    class B { public: virtual void f(); };

§ 11.7.2
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.

A consteval virtual function shall not override a virtual function that is not consteval. A consteval virtual function shall not be overridden by a virtual function that is not consteval.

### 11.7.3 Abstract classes

A class is an abstract class if it has at least one pure virtual function. [Note: An abstract class type cannot be used as a parameter or return type of a function being defined (9.3.3.5) or called (7.6.1.2), except as specified in 9.2.8.2. Further, an abstract class type cannot be used as the type of an explicit type conversion (7.6.1.8, 7.6.1.9, 7.6.1.10), 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]
would make class `circle` non-abstract and a definition of `circle::draw()` must be provided. — end example]

5 [Note: An abstract class can be derived from a class that is not abstract, and a pure virtual function may 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 (11.7.2) 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

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).

The following steps define the result of name lookup for a member name `f` in a class scope `C`.

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:

1 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. [Note: Looking up a name in an elaborated-type-specifier (6.5.4) or base-specifier (11.7), for instance, ignores all non-type declarations, while looking up a name in a nested-name-specifier (6.5.3) ignores function, variable, and enumerator declarations. As another example, looking up a name in a using-declaration (9.9) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name in the same scope. — end note] If the resulting declaration set is not empty, the subobject set contains `C` itself, and calculation is complete.

2 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)` into 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.

4 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:

```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:
struct A {
  int f();
};
struct B {
  int f();
};
struct C : A, B {
  int f() { return A::f() + B::f(); }
};
- end example
```

9 [Note: 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:
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 {
}
```

10 [Note: 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:
```
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(); }
```
As illustrated in Figure 6, the names declared in \texttt{V} and the left-hand instance of \texttt{W} are hidden by those in \texttt{B}, but the names declared in the right-hand instance of \texttt{W} are not hidden at all.

```cpp
void D::gloop()
{
x++;       // OK: \texttt{B::x} hides \texttt{V::x}
f();      // OK: \texttt{B::f()} hides \texttt{V::f()}
y++;       // error: \texttt{B::y} and \texttt{C's \texttt{W}::y}
g();      // error: \texttt{B::g()} and \texttt{C's \texttt{W}::g()}
}
```

— end example

11 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:

```cpp
struct V { }
struct A { }
struct B : A, virtual V { }
struct C : A, virtual V { }
struct D : B, C { }

void g()
{
    D d;
    B* pb = &d;
    A* pa = &d;      // error: ambiguous: \texttt{C's A} or \texttt{B's A}?
    V* pv = &d;      // OK: only one \texttt{V} subobject
}
```

— end example ]

12 [Note: Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (7.3.12, 7.6.1.4, 11.9.2). — end note ]

```cpp
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 \texttt{B2})
        int B1::* mpB1 = &D::i; // Unambiguous
        int D::* mpD = &D::i;   // Ambiguous conversion
    }
};
```

— end example

11.9 Member access control

A member of a class can be

\hspace{1cm} (1.1) private; that is, its name can be used only by members and friends of the class in which it is declared.

\hspace{1cm} (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 11.9.4).

\hspace{1cm} (1.3) public; that is, its name can be used anywhere without access restriction.

§ 11.9
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.\(^\text{11)}\)

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:

```cpp
class X {
    int a; // X::a is private by default
};

struct S {
    int a; // S::a is public by default
};
```

—end example]

Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions. [Note: Access control applies to names nominated by friend declarations (11.9.3) 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: 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]

[Note: 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.

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: This access also applies to implicit references to constructors, conversion functions, and destructors. — end note]

[Example:

```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 { }
```

\(^{11)}\) Access permissions are thus transitive and cumulative to nested and local classes.
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 (9.3.3.6) 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 13.9.1.

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:

```c++
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
};

--end example]
```

1. Member declarations can be labeled by an access-specifier (11.7):

   
   access-specifier : member-specificationopt

   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:

   ```c++
   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
   };
   --end example]
   ```

2. Any number of access specifiers is allowed and no particular order is required. [Example:

   ```c++
   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
   };
   --end example]
   ```

3. [Note: The effect of access control on the order of allocation of data members is specified in 7.6.9. —end note]

4. 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:

   ```c++
   struct S {
     class A;
     enum E : int;
     private:
     class A {};
     // error: cannot change access
   };
   ```

§ 11.9.1
enum E: int { e0 }; // error: cannot change access
};
—end example]  

[Note: 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:  
  class A { };  
  class B : private A { };  
  class C : public B {  
    A* p; // error: injected-class-name A is inaccessible  
    ::A* q; // OK  
  };  
—end example]  

11.9.2 Accessibility of base classes and base class members [class.access.base]  

1 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.  

2 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:  
  class B { /* ... */ };  
  class D1 : private B { /* ... */ };  
  class D2 : public B { /* ... */ };  
  class D3 : B { /* ... */ }; // B private by default  
  struct D4 : private B { /* ... */ };  
  struct D5 : public 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]  

3 [Note: 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 (7.3.11) and explicit casts (7.6.1.3, 7.6.1.8, 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,  
  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;  

112) 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.  

§ 11.9.2  288
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
::B* bp1 = this; // error: B is a private base class
::B* bp2 = (::B*)this; // OK with cast
bp2->mi = 3; // OK: access through a pointer to B.
}

— end note]

4 A base class B of N is accessible at R, if

(4.1) — an invented public member of B would be a public member of N, or

(4.2) — 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

(4.3) — 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

(4.4) — 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:

```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.
    }
};

— end example]

5 If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (7.3.11, 7.3.12). [Note: It follows that members and friends of a class X can implicitly convert an X* to a pointer to a private or protected immediate base class of X. — end note] The access to a member is affected by the class in which the member is named. This naming class is the class in which the member name was looked up and found. [Note: This class can be explicit, e.g., when a qualified-id is used, or implicit, e.g., when a class member access operator (7.6.1.4) is used (including cases where an implicit “this->” is added). If both a class member access operator and a qualified-id are used to name the member (as in p->T::m), the class naming the member is the class denoted by the nested-name-specifier of the qualified-id (that is, T). — end note] A member m is accessible at the point R when named in class N if

(5.1) — m as a member of N is public, or

(5.2) — m as a member of N is private, and R occurs in a member or friend of class N, or

(5.3) — m as a member of N is protected, and R occurs in a member or friend of class N, or in a member of a class P derived from N, where m as a member of P is public, private, or protected, or

(5.4) — there exists a base class B of N that is accessible at R, and m is accessible at R when named in class B.

[Example:

```cpp
class B;
class A {
    private:
    int i;
    friend void f(B*);
};
class B : public A { }
```
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: This requirement is in addition to the requirement that the member be accessible as named. —end note]

11.9.3 Friends [class.friend]

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: The following example illustrates the differences between members and friends:

```cpp
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:]

```cpp
class X {
    enum { a=100 }; // OK, Y is a friend of X
};

class Y {
    int v[X::a]; // error: X::a is private
};

—end example]
```
A class shall not be defined in a friend declaration. [Example:

```cpp
class A {
    friend class B { };  // error: cannot define class in friend declaration
};
—end example]
```

3 A friend declaration that does not declare a function shall have one of the following forms:

```cpp
friend elaborated-type-specifier;
friend simple-type-specifier;
friend typename-specifier;
```

[Note: A friend declaration may be the declaration in a `template-declaration` (13.1, 13.7.4). — end note] If the type specifier in a friend declaration designates a (possibly cv-qualified) class type, that class is declared as a friend; otherwise, the friend declaration is ignored. [Example:

```cpp
class C;
typedef C Ct;

class X1 {
    friend C;  // OK: class C is a friend
};

class X2 {
    friend Ct;  // OK: class C is a friend
    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
—end example]
```

4 A function first declared in a friend declaration has the linkage of the namespace of which it is a member (6.6, 9.8.1.2). Otherwise, the function retains its previous linkage (9.2.1).

5 When a friend declaration refers to an overloaded name or operator, only the function specified by the parameter types becomes a friend. A member function of a class X can be a friend of a class Y. [Example:

```cpp
class Y {
    friend char* X::foo(int);
    friend X::X(char);  // constructors can be friends
    friend X::~X();  // destructors can be friends
};
—end example]
```

6 A function can be defined in a friend declaration of a class if and only if the class is a non-local class (11.6), the function name is unqualified, and the function has namespace scope. [Example:

```cpp
class M {
    friend void f() {}  // definition of global f, a friend of M,
    // not the definition of a member function
};
—end example]
```

7 Such a function is implicitly an inline (9.2.7) 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 (6.5.1).

8 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:*

```cpp
class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p) {
        p->a++; // error: C is not a friend of A despite being a friend of a friend
    }
};

class D : public B {
    void f(A* p) {
        p->a++; // error: D is not a friend of A despite being derived from a friend
    }
};
```

—end example]*

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:*

```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
}
```

—end example]*

### 11.9.4 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 (11.9.2). 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 (7.6.2.1), the nested-name-specifier shall denote C or a class derived from C. All other accesses involve a (possibly implicit) object expression (7.6.1.4). In this case, the class of the object expression shall be C or a class derived from C. [*Example:*

```cpp
113) This additional check does not apply to other members, e.g., static data members or enumerator member constants.
```
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 a D2, even though naming class is B)
    int B::* pm1_B = &B::i; // error
    int B::* pm1_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::* pm1_B = &B::i; // error
    int B::* pm1_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]

11.9.5 Access to virtual functions [class.access.virt]

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:

class B {
    public:
        virtual int f();
};

class D : public B {
    private:
        int f();
};

void f() {
    D d;
    B* pb = &d;
    D* pd = &d;
}
pb->f();  // OK: B::f() is public, D::f() is invoked
pd->f();  // error: D::f() is private
}

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.

11.9.6 Multiple access

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:

```cpp
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]

11.9.7 Nested classes

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:

```cpp
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

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.

An object of class type (or array thereof) can be explicitly initialized; see 11.10.1 and 11.10.2.

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 9.3.3.4. [Note: Destructors for the array elements are called in reverse order of their construction. — end
note]

11.10.1 Explicit initialization

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:

```cpp
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; // initialized by calling complex()
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: Overloading of the assignment operator (12.6.2.1) 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 9.4.4. [Example:
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,

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: 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 9.4.1. —end note]

3 [Note: 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: 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]

11.10.2 Initializing bases and members

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
mem-initializer-list:
  mem-initializer ...opt
  mem-initializer-list , mem-initializer ...opt
mem-initializer:
  mem-initializer-id ( expression-listopt )
  mem-initializer-id braced-init-list
mem-initializer-id:
  class-or-decltype
  identifier

2 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: 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 may 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.
A mem-initializer-list can initialize a base class using any class-or-decltype that denotes that base class type. [Example:

```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]

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:

```c
struct A { A(); }
struct B: public virtual A { }
struct C: public A, public B { C(); }
C::C(): A() { }         // error: which A?
```
—end example]

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.

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:

```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]

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:

```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: 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.

A temporary expression bound to a reference member in a mem-initializer is ill-formed. [Example:

```c
struct A {
    A() : v(42) { } // error
    const int& v;
};
```
—end example]
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 (11.5.1), no initialization is performed;

(9.3) — otherwise, the entity is default-initialized (9.4).

[Note: An abstract class (11.7.3) is never a most derived class, thus its constructors never initialize virtual base classes, therefore the corresponding `mem-initializers` may be omitted. — end note] An attempt to initialize more than one non-static data member of a union renders the program ill-formed. [Note: 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:

```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
};
```

— end example]

10 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: Given

```c
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]

11 A temporary expression bound to a reference member from a default member initializer is ill-formed. [Example:

```c
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
```

§ 11.10.2
In a non-delegating constructor, the destructor for each potentially constructed subobject of class type is potentially invoked (11.4.6). [Note: 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: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. —end note]

Example:

```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();
    C(int);
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }
```

V v(1); // use V(int)
A a(2);  // use V(int)
B b(3);  // use V()
C c(4);  // use V()

—end example]

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:

```cpp
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: 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, 11.7.2) can be called for an object under construction. Similarly, an object under construction can be the operand of the typeid operator (7.6.1.7) or of a dynamic_cast (7.6.1.6). 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:

```cpp
class A {
public:
  A(int);
};

class B : public A {
  int j;
public:
  int f();
  B() : A(f()), j(f()) { }  // well-defined: bases are all initialized
};

class C {
public:
  C(int);
};

class D : public B, C {
  int i;
public:
  D() : C(f()), i(f()) { }  // well-defined: bases are all initialized
};
—end example]

[Note: 11.10.4 describes the result of virtual function calls, typeid and dynamic_casts during construction for the well-defined cases; that is, describes the polymorphic behavior of an object under construction. — end note]

A mem-initializer followed by an ellipsis is a pack expansion (13.7.3) that initializes the base classes specified by a pack expansion in the base-specifier-list for the class. [Example:

```cpp
template<class... Mixins>
class X : public Mixins... {
public:
  X(const Mixins&... mixins) : Mixins(mixins)... { }
};
—end example]

11.10.3 Initialization by inherited constructor [class.inhctor.init]

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:

```cpp
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

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
  ~Log() { std::clog << "Destroying wrapper" << std::endl; }
};

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]

2 If the constructor was inherited from multiple base class subobjects of type B, the program is ill-formed.

[Example:

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] 3

When an object is initialized by an inherited constructor, initialization of the object is complete when the initialization of all subobjects is complete.

11.10.4 Construction and destruction [class.cdtor]

1 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:

```cpp
struct X { int i; };
struct Y : X { Y(); };
struct A { int a; };
struct B : public A { int j; Y y; };
```

```cpp
extern B bobj;
B* pb = &bobj; // OK
int* p1 = &bobj.a;
int* p2 = &bobj.y.i;
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 {};
struct Y {
    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:

```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 << '\n'; // value of cobj.c is unspecified
}
```

§ 11.10.4 301
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

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:]

```c
struct A { }
struct B : virtual A { }
struct C : B { }
struct D : virtual A { D(A*); }
struct X { X(A*); }

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
    X(this) {} // defined: upon construction of X, C/B/D/A sublattice is fully constructed
};

—end example
```

Member functions, including virtual functions (11.7.2), can be called during construction or destruction (11.10.2). 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 (7.6.1.4) 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:]

```c
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
```
The typeid operator (7.6.1.7) can be used during construction or destruction (11.10.2). 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 (7.6.1.6) can be used during construction or destruction (11.10.2). 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. [Example:

```cpp
struct V {
  virtual void f();
};

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); // type_info for B
  typeid(*v);   // well-defined: *v has type V, a base of B yields type_info for B
  typeid(a);   // undefined behavior: a not a base of B
  dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B results in B*
  dynamic_cast<B*>(a); // undefined behavior: a has type A*, A not a base of B
}
```

— end example]
as the function return type, the copy/move operation can be omitted by constructing the object directly into the function call’s return object.

(1.2) — 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.

(1.3) — 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.

(1.4) — 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: 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: Copy elision might be performed if the same expression is evaluated in another context. — end note]

2 [Example:
```cpp
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};

Thing f() {
  Thing t;
  return t;
}

Thing t2 = f();

struct A {
  void *p;
  constexpr A(): p(this) {}
};

constexpr A g() {
  A loc;
  return loc;
}

constexpr A a; // well-formed, a.p points to a
constexpr A b = g(); // error: b.p would be dangling (7.7)

void h() {
  A c = g(); // well-formed, c.p may point to c or to an ephemeral temporary
}
```
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 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]

3 An 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 might be used instead of a copy operation:
If the expression in a return (8.7.3) or co_return (8.7.4) statement is a (possibly parenthesized) id-expression that names an implicitly movable entity declared in the body or parameter-declaration-clause of the innermost enclosing function or lambda-expression, or

— if the operand of a throw-expression (7.6.18) is a (possibly parenthesized) id-expression that names an implicitly movable entity whose scope does not extend beyond the compound-statement of the innermost try-block or function-try-block (if any) whose compound-statement or ctor-initializer encloses the throw-expression,

overload resolution to select the constructor for the copy or the 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. [Note: This two-stage overload resolution must be performed regardless of whether copy elision will occur. It determines the constructor or the return_value overload to be called if elision is not performed, and the selected constructor or return_value overload must be accessible even if the call is elided. —end note]

4 [Example:
   
   ```
   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]

5 [Example:
   
   ```
   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
   ```
   
   § 11.10.5 305
11.11 Comparisons

11.11.1 Defaulted comparison operator functions

A defaulted comparison operator function (12.6.2) for some class $C$ shall be a non-template function that is

(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 $&$, or

(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.

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 (11.5.1).

A binary operator expression $a @ b$ is usable if either

(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

(3.2) 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 (9.2.5) and no overload resolution performed when determining whether to delete the function results in a usable candidate that is a non-constexpr function. [Note: 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]

A defaulted comparison 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 $operator==$.

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: 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 may differ from the implicit exception specification of the three-way comparison operator function. — end note] [Example:

```cpp
template<
type 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: The $ operator function is declared implicitly even if the defaulted three-way comparison operator function is defined as deleted. — end note]

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 (12.4.3.1), class member access expressions (7.6.1.4), and array subscript expressions (7.6.1.1) applied to \( x \).

### 11.11.2 Equality operator

1 A defaulted equality operator function (12.6.2) shall have a declared return type \( \text{bool} \).

2 A defaulted \( \text{==} \) 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 \( C \), \( x_i \text{==} x_i \) is usable (11.11.1).

3 The return value \( V \) of a defaulted \( \text{==} \) 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 \( x_i \text{==} y_i \) yields a result value which, when contextually converted to \( \text{bool} \), yields \( \text{false} \). If no such index exists, \( V \) is \( \text{true} \). Otherwise, \( V \) is \( \text{false} \).

4 [Example:

\[
\text{struct D } \\
\quad \{ \\
\quad \quad \text{int} \ i; \\
\quad \quad \text{friend bool operator==}(\text{const D&} \ x, \text{const D&} \ y) = \text{default}; \\
\quad \quad \quad \quad \text{// OK, returns } x.i \text{==} y.i \\
\quad \}; \\
\quad \text{— end example}
\]

### 11.11.3 Three-way comparison

1 The synthesized three-way comparison of type \( R \) (17.11.2) of glvalues \( a \) and \( b \) of the same type is defined as follows:

\[
\begin{align*}
(1.1) & \quad \text{If } a \text{ <=>} b \text{ is usable (11.11.1), } \text{static_cast}<R>(a \text{ <=>} b). \\
(1.2) & \quad \text{Otherwise, if overload resolution for } a \text{ <=>} b \text{ is performed and finds at least one viable candidate, the synthesized three-way comparison is not defined.} \\
(1.3) & \quad \text{Otherwise, if } R \text{ is not a comparison category type, or either the expression } a \text{ ==} b \text{ or the expression } a < b \text{ is not usable, the synthesized three-way comparison is not defined.} \\
(1.4) & \quad \text{Otherwise, if } R \text{ is } \text{strong\_ordering}, \text{ then} \\
& \quad \quad a \text{ ==} b \ ? \text{strong\_ordering::equal} : \\
& \quad \quad a < b \ ? \text{strong\_ordering::less} : \\
& \quad \quad \quad \text{strong\_ordering::greater} \\
(1.5) & \quad \text{Otherwise, if } R \text{ is } \text{weak\_ordering}, \text{ then} \\
& \quad \quad a \text{ ==} b \ ? \text{weak\_ordering::equivalent} : \\
& \quad \quad a < b \ ? \text{weak\_ordering::less} : \\
& \quad \quad \quad \text{weak\_ordering::greater} \\
(1.6) & \quad \text{Otherwise (when } R \text{ is } \text{partial\_ordering}, \text{)} \\
& \quad \quad a \text{ ==} b \ ? \text{partial\_ordering::equivalent} : \\
& \quad \quad a < b \ ? \text{partial\_ordering::less} : \\
& \quad \quad b < a \ ? \text{partial\_ordering::greater} : \\
& \quad \quad \quad \text{partial\_ordering::unordered}
\end{align*}
\]

[Note: A synthesized three-way comparison may be ill-formed if overload resolution finds usable candidates that do not otherwise meet the requirements implied by the defined expression. — end note]

2 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 \( C \).

\[
\begin{align*}
(2.1) & \quad \text{If } R \text{ is } \text{auto}, \text{ then let } cv0, R_i \text{ be the type of the expression } x_i \text{ <=>} x_i. \text{ The operator function is defined as deleted if that expression is not usable or if } R_i \text{ is not a comparison category type (17.11.2.1) for any } i. \text{ The return type is deduced as the common comparison type (see below) of } R_0, R_1, \ldots, R_{n-1}. \\
(2.2) & \quad \text{Otherwise, } R \text{ shall not contain a placeholder type. If the synthesized three-way comparison of type } R \text{ between any objects } x_i \text{ and } x_i \text{ is not defined, the operator function is defined as deleted.}
\end{align*}
\]
The return value \( V \) 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 \( v_i \neq 0 \), contextually converted to \( \text{bool} \), yields \( \text{true} \); \( V \) is a copy of \( v_i \). If no such index exists, \( V \) is \( \text{static_cast}<\text{R}>(\text{std::strong_ordering::equal}) \).

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. \( U \) is \( \text{std::partial_ordering} \) \((17.11.2.2)\).
2. \( U \) is \( \text{std::weak_ordering} \) \((17.11.2.3)\).
3. \( U \) is \( \text{std::strong_ordering} \) \((17.11.2.4)\). [Note: In particular, this is the result when \( n \) is 0. —end note]

### Secondary comparison operators

A secondary comparison operator is a relational operator \((7.6.9)\) or the \(!= \) operator. A defaulted operator function \((12.6.2)\) 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.

### Free store

Any allocation function for a class \( T \) is a static member (even if not explicitly declared \( \text{static} \)).

When an object is deleted with a \( \text{delete-expression} \) \((7.6.2.8)\), a deallocation function (\( \text{operator delete}() \) for non-array objects or \( \text{operator delete[]}() \) for arrays) is (implicitly) called to reclaim the storage occupied by the object \((6.7.5.4.2)\).

Class-specific deallocation function lookup is a part of general deallocation function lookup \((7.6.2.8)\) and occurs as follows. If the \( \text{delete-expression} \) 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 (11.4.6). 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 (7.6.2.8) 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.

Any deallocation function for a class X is a static member (even if not explicitly declared static). [Example:

```cpp
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]

Since member allocation and deallocation functions are static they cannot be virtual. [Note: 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,

```cpp
struct B {
    virtual ~B();
    void operator delete(void*, std::size_t);
};

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;     // 1: uses D::operator delete(void*)
    bp = new E;
    delete bp;     // 2: 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: 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,

```cpp
struct B {
    virtual ~B();
    void operator delete[](void*, std::size_t);
};

struct D : B {
    void operator delete[](void*, std::size_t);
};
```

115) A similar provision is not needed for the array version of operator delete because 7.6.2.8 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.
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. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. [Example: 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: 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:

```c++
double abs(double);
int abs(int);

abs(1); // calls abs(int);
abs(1.0); // calls abs(double);
```

—end example]

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: 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). —end note]

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 (9.3.3.5), and the same trailing requires-clause (if any) cannot be overloaded if any of them is a static member function declaration (11.4.8). 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 (12.4.1) 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: 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
    void g() const volatile; // OK: no static
};
```

—end example]

(2.3) — Member function declarations with the same name, the same parameter-type-list (9.3.3.5), 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 (9.3.3.5). [Example:
class Y {
    void h() &;
    void h() const &;  // OK
    void h() &
    void i() &;
    void i() const;   // error: prior declaration of i has a ref-qualifier
};

—end example

3 [Note: As specified in 9.3.3.5, function declarations that have equivalent parameter declarations and requires-clauses, if any (13.5.2), 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 (9.2.3). [Example:

    typedef int Int;
    void f(int i);
    void f(Int i);   // OK: redeclaration of f(int)
    void f(int i) { /* ... */ }  // error: redefinition of f(int)
    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:

    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 (9.3.3.5). Only the second and subsequent array dimensions are significant in parameter types (9.3.3.4). [Example:

    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 (9.3.3.5). [Example:

    void h(int());
    void h(int (*));  // redeclaration of h(int())
    void h(int x());
    void h(int (*x)()) { }  // definition of h(int())
    void h(int (*x)()) { }  // error: redefinition of h(int())

—end example]

(3.4) — 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:

    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: 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()?
}
```

— end example] — end note]  

12.3 Declaration matching [over.dcl]

1 Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (12.2) and equivalent (13.7.6.1) trailing `requires-clauses`, if any (9.3). [Note: Since a `constraint-expression` is an unevaluated operand, equivalence compares the expressions without evaluating them. [Example:

```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
};
```

— end example] — end note] 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:

```c
struct B {
    int f(int);
};

struct D : B {
    int f(const char*);
};
```

Here `D::f(const char*)` hides `B::f(int)` rather than overloading it.

```c
void h(D* pd) {
    pd->f();             // 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:

```c
void f(const char*);
void g() {
    extern void f(int);
```
3 Different versions of an overloaded member function can be given different access rules. [Example:

```cpp
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

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:
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]

Overload resolution selects the function to call in seven distinct contexts within the language:

1. invocation of a function named in the function call syntax (12.4.1.1.1);
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 (12.4.1.1.2);
3. invocation of the operator referenced in an expression (12.4.1.2);
4. invocation of a constructor for default- or direct-initialization (9.4) of a class object (12.4.1.3);
5. invocation of a user-defined conversion for copy-initialization (9.4) of a class object (12.4.1.4);
6. invocation of a conversion function for initialization of an object of a non-class type from an expression
of class type (12.4.1.5); and
7. invocation of a conversion function for conversion in which a reference (9.4.3) will be directly bound
(12.4.1.6).

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:

1. 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 (12.4.2).
2. Then the best viable function is selected based on the implicit conversion sequences (12.4.3.1) 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.

4 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.

12.4.1 Candidate functions and argument lists  

The subclauses of 12.4.1 describe the set of candidate functions and the argument list submitted to overload resolution in each context in which overload resolution is used. The source transformations and constructions defined in these subclauses are only for the purpose of describing the overload resolution process. An implementation is not required to use such transformations and constructions.

2 The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra first parameter, called the implicit object parameter, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

3 Similarly, when appropriate, the context can construct an argument list that contains an implied object argument as the first argument in the list to denote the object to be operated on.

4 For non-static member functions, the type of the implicit object parameter is

\[ \begin{align*}
&\text{(4.1)} \quad \text{"lvalue reference to } cv \ X \text{" for functions declared without a ref-qualifier or with the } \& \text{ ref-qualifier} \\
&\text{(4.2)} \quad \text{"rvalue reference to } cv \ X \text{" for functions declared with the } && \text{ ref-qualifier}
\end{align*} \]

where \( X \) is the class of which the function is a member and \( cv \) is the cv-qualification on the member function declaration. [Example: For a \texttt{const} member function of class \( X \), the extra parameter is assumed to have type “reference to \texttt{const} \ X \”. — end example] For conversion functions, the function is considered to be a member of the class of the implied object argument for the purpose of defining the type of the implicit object parameter. For non-conversion functions introduced by a \textit{using-declaration} into a derived class, the function is considered to be a member of the derived class for the purpose of defining the type of the implicit object parameter. For static member functions, the implicit object parameter is considered to match any object (since if the function is selected, the object is discarded). [Note: No actual type is established for the implicit object parameter of a static member function, and no attempt will be made to determine a conversion sequence for that parameter (12.4.3). — end note]

5 During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since no user-defined conversions can be applied to achieve a type match with it. For non-static member functions declared without a \textit{ref-qualifier}, even if the implicit object parameter is not \texttt{const}-qualified, an \textit{rvalue} can be bound to the parameter as long as in all other respects the argument can be converted to the type of the implicit object parameter. [Note: The fact that such an argument is an \textit{rvalue} does not affect the ranking of implicit conversion sequences (12.4.3.2). — end note]

6 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 (12.4.3, 12.4.3.1). [Example:

```cpp
class T {
public:
    T();
};

class C : T {
public:
    C(int);
};
T a = 1; // error: no viable conversion (T(C(1))) not considered
```

— end example]

7 In each case where a candidate is a function template, candidate function template specializations are generated using template argument deduction (13.10.3, 13.10.2). If a constructor template or conversion function template has an \textit{explicit-specifier} whose \textit{constant-expression} is value-dependent (13.8.2), template
argument deduction is performed first and then, if the context requires a candidate that is not explicit and the generated specialization is explicit (9.2.2), 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.

8 A defaulted move special member function (11.4.4.2, 11.4.5) that is defined as deleted is excluded from the set of candidate functions in all contexts. A constructor inherited from class type C (11.10.3) 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:

```c++
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;
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&&(); };  // calls #4
B b3 = C();
—end example]

12.4.1.1 Function call syntax [over.match.call]

1 In a function call (7.6.1.2)

\[
\text{postfix-expression} \ (\text{expression-list}_{\text{opt}})
\]

if the \textit{postfix-expression} names at least one function or function template, overload resolution is applied as specified in 12.4.1.1.1. If the \textit{postfix-expression} denotes an object of class type, overload resolution is applied as specified in 12.4.1.1.2.

2 If the \textit{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: The resolution of the address of an overload set in other contexts is described in 12.5. —end note]

12.4.1.1.1 Call to named function [over.call.func]

1 Of interest in 12.4.1.1.1 are only those function calls in which the \textit{postfix-expression} ultimately contains a name that denotes one or more functions that might be called. Such a \textit{postfix-expression}, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

\[
\text{postfix-expression:}
\text{    postfix-expression . id-expression}
\text{    postfix-expression \texttt{\rightarrow} id-expression}
\text{    primary-expression}
\]

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2 In qualified function calls, the name to be resolved is an \textit{id-expression} and is preceded by an \texttt{\rightarrow} or \texttt{.} operator. Since the construct \texttt{A->B} is generally equivalent to \texttt{(*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 \texttt{.} operator. Furthermore, Clause 12 assumes that the \textit{postfix-expression} that is the left operand of the \texttt{.} operator.

117) 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 (9.3.3.5) are treated equivalently for the remainder of overload resolution.

\[\text{§ 12.4.1.1.1} \]
operator has type “cv T” where T denotes a class. \(^{118}\) 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 (12.4.1).

In unqualified function calls, the name is not qualified by an \( \rightarrow \) 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 (11.4.2.1) 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 of type T becomes the implied object argument. \(^{119}\) 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.

12.4.1.1.2 Call to object of class type \([\text{over.call.object}]\)

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

\[
\text{operator conversion-type-id ( ) cv-qualifier-seq_opt ref-qualifier_opt noexcept-specifier_opt attribute-specifier-seq_opt ;}
\]

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 \((P_1, \ldots, P_n)\) returning R”, or the type “reference to pointer to function of \((P_1, \ldots, P_n)\) returning R”, or the type “reference to function of \((P_1, \ldots, P_n)\) returning R”, a surrogate call function with the unique name call-function and having the form

\[
\text{R call-function ( conversion-type-id F, P_1 a_1, \ldots, P_n a_n) \{ return F (a_1, \ldots, a_n); \}}
\]

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. \(^{120}\)

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). \([\text{Note: When comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. — end note}]\)

Example:

```cpp
int f1(int);
int f2(float);
typedef int (*fp1)(int);
typedef 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
```

\(^{118}\) Note that cv-qualifiers on the type of objects are significant in overload resolution for both gvalue and prvalue objects.

\(^{119}\) An implied object argument must be 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.

\(^{120}\) 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.
12.4.1.2 Operators in expressions

1 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: 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:

```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
    int I = 1 + 1 ; // always evaluates to 2 even if class or enumeration types exist
} // that would perform the operation.
```
— end example]

2 If either operand has a type that is a class or an enumeration, a user-defined operator function might 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).

### 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>12.6.1</td>
<td>a@</td>
<td>(a).operator@ ()</td>
<td>operator@(a)</td>
</tr>
<tr>
<td>12.6.2</td>
<td>a@b</td>
<td>(a).operator@ (b)</td>
<td>operator@(a, b)</td>
</tr>
<tr>
<td>12.6.2.1</td>
<td>a=b</td>
<td>(a).operator= (b)</td>
<td></td>
</tr>
<tr>
<td>12.6.4</td>
<td>a[b]</td>
<td>(a).operator[] (b)</td>
<td></td>
</tr>
<tr>
<td>12.6.5</td>
<td>a-&gt;</td>
<td>(a).operator-&gt;()</td>
<td></td>
</tr>
<tr>
<td>12.6.6</td>
<td>a@</td>
<td>(a).operator@ (0)</td>
<td>operator@(a, 0)</td>
</tr>
</tbody>
</table>

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:

(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@ (12.4.1.1.1); 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 (6.5.2) 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
— accept operand types to which the given operand or operands can be converted according to 12.4.3.1, and

— do not have the same parameter-type-list as any non-member candidate that is not a function template specialization.

— The rewritten candidate set is determined as follows:

— For the relational (7.6.9) operators, the rewritten candidates include all non-rewritten candidates for the expression \( x \leftrightarrow y \).

— 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 \( y \leftrightarrow x \).

— For the \(!=\) operator (7.6.10), the rewritten candidates include all non-rewritten candidates for the expression \( x == y \).

— 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 \( y == x \).

— For all other operators, the rewritten candidate set is empty.

[Note: 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:

— no temporaries are introduced to hold the left operand, and

— 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 \(@\).

7 The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 12.4.2 and 12.4.3.121 [Example:

```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]
```

8 If a rewritten \(\leftrightarrow\) candidate is selected by overload resolution for an operator \(@\), \(x \leftrightarrow y\) is interpreted as \(0 \leftrightarrow (y \leftrightarrow x)\) if the selected candidate is a synthesized candidate with reversed order of parameters, or \((x \leftrightarrow y) \leftrightarrow 0\) otherwise, using the selected rewritten \(\leftrightarrow\) candidate. Rewritten candidates for the operator \(@\) are not considered in the context of the resulting expression.

9 If a rewritten \(==\) candidate is selected by overload resolution for an operator \(@\), its return type shall be \(cv\) bool, and \(x \leftrightarrow y\) is interpreted as:

\[\begin{align*}
((\@ \text{ is } !=) \text{ and the selected candidate is a synthesized candidate with reversed order of parameters, } \neg(x == z)), \\
(\neg(\@ \text{ is } !=), \neg(x == z))
\end{align*}\]

in each case using the selected rewritten \(==\) candidate.

121) If the set of candidate functions is empty, overload resolution is unsuccessful.
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 (12.4.3.2) is not applied. Then the operator is treated as the corresponding built-in operator and interpreted according to 7.6. [Example:

```cpp
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.122

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: 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:

```cpp
struct A { };  
void operator + (A, A);

struct B {
    void operator + (B);
    void f();
};

A a;

void B::f() {
    operator+ (a,a);       // error: global operator hidden by member
    a + a;                  // OK: calls global operator+
}
—end note]

12.4.1.3 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 (11.4.7.1) of that class. The argument list is the expression-list or assignment-expression of the initializer.

12.4.1.4 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: 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:

122) If the value returned by the `operator->` function has class type, this may result in selecting and calling another `operator->` function. The process repeats until an `operator->` function returns a value of non-class type.
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 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: This argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. — end note]

**12.4.1.5 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 (12.4.3.1.1) 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 (7.3.5) 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: This argument will be compared against the implicit object parameter of the conversion functions. — end note]

**12.4.1.6 Initialization by conversion function for direct reference binding**

Under the conditions specified in 9.4.3, 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 cv1 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 lvalue reference or an rvalue reference to function) or “cv2 T2” or “rvalue reference to cv2 T2” (when initializing an rvalue reference or an lvalue reference to function), where “cv1 T” is reference-compatible (9.4.3) with “cv2 T2”, are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within S and yield type “lvalue reference to cv2 T2” (when initializing an lvalue reference or an rvalue reference to function) or “rvalue reference to cv2 T2” (when initializing an rvalue reference or an lvalue reference to function), where T2 is the same type as T or can be converted to type T with a qualification conversion (7.3.5), are also candidate functions.

2. The argument list has one argument, which is the initializer expression. [Note: This argument will be compared against the implicit object parameter of the conversion functions. — end note]

**12.4.1.7 Initialization by list-initialization**

When objects of non-aggregate class type T are list-initialized such that 9.4.4 specifies that overload resolution is performed according to the rules in this subclause or when forming a list-initialization sequence according to 12.4.3.1.5, 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 (9.4.4) of the class T and the argument list consists of the initializer list as a single argument.
12.4.1.8 Class template argument deduction

When resolving a placeholder for a deduced class type (9.2.8.6) 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:
   - The template parameters are the template parameters of C followed by the template parameters (including default template arguments) of the constructor, if any.
   - The types of the function parameters are those of the constructor.
   - 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:
   - The template parameters, if any, and function parameters are those of the deduction-guide.
   - 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 (9.4.1) 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 \( x_i \) (9.4.1) if

1. 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
2. each non-trailing aggregate element that is a pack expansion is assumed to correspond to no elements of the initializer list, and
3. 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(T₁, \ldots, Tₙ), where

1. 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
2. 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 (9.2.8.2) where the template-name names an alias template Λ, the defining-type-id of Λ must be of the form

\[
\text{typename}_{opt} \text{ nested-name-specifier}_{opt} \text{ template}_{opt} \text{ simple-template-id}
\]

as specified in 9.2.8.2. The guides of Λ 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
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 \( A \) (13.8.2.1), \( AA<T> \) matches the partial specialization.

Initialization and overload resolution are performed as described in 9.4 and 12.4.1.3, 12.4.1.4, or 12.4.1.7 (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 12.4.1.7 (considering initializer-list constructors) is omitted if the initializer list consists of a single expression of type \( \text{cv } U \), 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:]

```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*>  

template <typename T>
struct S {
    T x;
    T y;
};

template <typename T>
struct C {
    S<T> s;
    T t;
};

template <typename T>
struct D {
    S<int> s;
    T t;
};

C c1 = {1, 2};       // error: deduction failed
C c2 = {1, 2, 3};   // error: deduction failed
C c3 = {{1u, 2u}, 3};  // OK, deduces C<int>

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>

E e2 = {1, 2, 3};      // OK, deduces E<int>

E e3 = {1, 2, 3, 4};   // error: conflicting types deduced; operator Y not considered

—end example

6 [Example:

    template <class T, class U> struct C {
        C(T, U); // #1
    };

§ 12.4.1.8
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> requires deduces_A<C<T, std::type_identity_t<U>>>  
auto f2_prime(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>>> &&
                           deduces_B<C<V *, std::type_identity_t<U>>>  
auto f2_prime_for_B(V *, U) -> C<V *, std::type_identity_t<U>>;

—end example

12.4.2 Viable functions

From the set of candidate functions constructed for a given context (12.4.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences and associated constraints (13.5.2) for the best fit (12.4.3). The selection of viable functions considers associated constraints, if any, and relationships between arguments and function parameters other than the ranking of conversion sequences.
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 (9.3.3.5). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to “match the ellipsis” (12.4.3.1.3).

(2.3) A candidate function having more than \( m \) parameters is viable only if all parameters following the \( m \textsuperscript{th} \) have default arguments (9.3.3.6). For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly \( m \) parameters.

Second, for a function to be viable, if it has associated constraints (13.5.2), those constraints shall be satisfied (13.5.1).

Third, for \( F \) to be a viable function, there shall exist for each argument an implicit conversion sequence (12.4.3.1) 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-\texttt{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 12.4.3.1.4).

**12.4.3 Best viable function**

1 Define \( \text{ICS}_i(F) \) as follows:

(1.1) If \( F \) is a static member function, \( \text{ICS}_1(F) \) is defined such that \( \text{ICS}_1(F) \) is neither better nor worse than \( \text{ICS}_1(G) \) for any function \( G \), and, symmetrically, \( \text{ICS}_1(G) \) is neither better nor worse than \( \text{ICS}_1(F) \),\(^{123}\) otherwise,

(1.2) let \( \text{ICS}_i(F) \) denote the implicit conversion sequence that converts the \( i \text{-th} \) argument in the list to the type of the \( i \text{-th} \) parameter of viable function \( F \). 12.4.3.1 defines the implicit conversion sequences and 12.4.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

2 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 \( i \), \( \text{ICS}_i(F_1) \) is not a worse conversion sequence than \( \text{ICS}_i(F_2) \), and then

(2.1) for some argument \( j \), \( \text{ICS}_j(F_1) \) is a better conversion sequence than \( \text{ICS}_j(F_2) \), or, if not that,

(2.2) the context is an initialization by user-defined conversion (see 9.4, 12.4.1.5, and 12.4.1.6) 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:

```
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
```

— end example] or, if not that,

(2.3) the context is an initialization by conversion function for direct reference binding (12.4.1.6) 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:

```
template <class T> struct A {
    operator T&(); // #1
    operator T&&(); // #2
};
typedef int Fn();
A<Fn> a;
```

\(^{123}\) 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.
— end example] or, if not that,

(2.4) — F1 is not a function template specialization and F2 is a function template specialization, or, if not that,

(2.5) — F1 and F2 are function template specializations, and the function template for F1 is more specialized than the template for F2 according to the partial ordering rules described in 13.7.6.2, or, if not that,

(2.6) — F1 and F2 are non-template functions with the same parameter-type-lists, and F1 is more constrained than F2 according to the partial ordering of constraints described in 13.5.4, or if not that,

(2.7) — F1 is a constructor for a class D, F2 is a constructor for a base class B of D, and for all arguments the corresponding parameters of F1 and F2 have the same type. [Example:

```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 (12.4.1.2) and F1 is not [Example:

```cpp
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:

```cpp
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 (12.4.1.8) and F2 is not, or, if not that,

(2.11) — F1 is the copy deduction candidate (12.4.1.8) 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:

```cpp
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. [Example:

```cpp
define Fcn(const int*, short);
define 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:

```cpp
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]
```

12.4.3.1 Implicit conversion sequences
[over.best.ics]

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, 9.4.3).

124) 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.

§ 12.4.3.1 328
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. \[Note: \] 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

A well-formed implicit conversion sequence is one of the following forms:

1. A standard conversion sequence (12.4.3.1.1),
2. A user-defined conversion sequence (12.4.3.1.2), or
3. An ellipsis conversion sequence (12.4.3.1.3).

4. However, if the target is
   1. the first parameter of a constructor or
   2. the implicit object parameter of a user-defined conversion function and the constructor or user-defined conversion function is a candidate by
   3. 12.4.1.3, when the argument is the temporary in the second step of a class copy-initialization,
   4. 12.4.1.4, 12.4.1.5, or 12.4.1.6 (in all cases), or
   5. the second phase of 12.4.1.7 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 \(X\), and the conversion is to \(X\) or reference to \(cv\ X\),

user-defined conversion sequences are not considered. \[Note: \] These rules prevent more than one user-defined conversion from being applied during overload resolution, thereby avoiding infinite recursion. — end note

Example:

```
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(); };  
B b;  
X x{b});  // error: B::operator X() is not a candidate
```

For the case where the parameter type is a reference, see 12.4.3.1.4.

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. \[Note: \] When the parameter has a class type, this is a conceptual conversion defined for the purposes of Clause 12; the actual initialization 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. \[Example: \] A parameter of type \(A\) can be initialized from an argument of type \(const A\). The implicit conversion sequence for that case is the identity sequence; it contains no “conversion” from \(const A\) to \(A\). — end example] 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. \[Note: \] 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 (12.4.3.1.1).

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 (12.4.3.1.1).

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 12.4.3.2, the ambiguous conversion sequence is treated as a user-defined conversion sequence that is indistinguishable from any other user-defined conversion sequence. [Note: This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters.]

[Example:

```cpp
class B;
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.

The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

12.4.3.1.1 Standard conversion sequences [over.ics.scs]

Table 16 summarizes the conversions defined in 7.3 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. [Note: 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: 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 (12.4.3.2). The rank of a conversion sequence is determined by considering the rank of each of the conversion in the sequence and the rank of any reference binding (12.4.3.1.4). 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 [tab:over.ics.scs]
12.4.3.1.2 User-defined conversion sequences

1 A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (11.4.7) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (11.4.7.1), the initial standard conversion sequence converts the source type to the type required by the argument of the constructor. If the user-defined conversion is specified by a conversion function (11.4.7.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the 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 12.4.3 and 12.4.3.1).

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.

12.4.3.1.3 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 7.6.1.2).

12.4.3.1.4 Reference binding

1 When a parameter of reference type binds directly (9.4.3) 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 (12.4.3.1). [Example:

```c
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 (12.4.3.1.2), 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 12.4.3.1. 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 12.4.1, an implicit 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: 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 9.4.3). —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: 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 (9.4.3). —end example]
12.4.3.1.5 List-initialization sequence

1 When an argument is an initializer list (9.4.4), 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 (9.4.1), in which case the implicit conversion sequence is a user-defined conversion sequence whose second standard conversion sequence is an identity conversion. [Note: 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 (9.4.4).] [Example:

```
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
    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 (9.4.2), 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:

```
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.

---

125) Since there are no parameters of array type, this will only occur as the referenced type of a reference parameter.
Otherwise, if the parameter is a non-aggregate class `X` and overload resolution per 12.4.1.7 chooses a single best constructor `C` of `X` to perform the initialization of an object of type `X` from the argument initializer list:

(7.1) — 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`.

(7.2) — 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 12.4.3.1. [Example:

```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"))))
```

—end example]

Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (9.4.1), the implicit conversion sequence is a user-defined conversion sequence with the second standard conversion sequence an identity conversion. [Example:

```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
```

—end example]

9 Otherwise, if the parameter is a reference, see 12.4.3.1.4. [Note: The rules in this subclause will apply for initializing the underlying temporary for the reference. — end note] [Example:

```cpp
struct A {
    int m1;
    double m2;
};
void f(const A&);
```
f( {'a', 'b'} ); // OK: f(A(int,double)) user-defined conversion
f( {1.0} ); // error: narrowing

void g(const double &); g({1}); // same conversion as int to double
— end example]

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:

    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:

    void f(int);
    f( {} ); // OK: identity conversion
    — end example]

In all cases other than those enumerated above, no conversion is possible.

12.4.3.2 Ranking implicit conversion sequences
[over.ics.rank]

1 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 12.4.3.1)
   — a standard conversion sequence (12.4.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and
   — a user-defined conversion sequence (12.4.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (12.4.3.1.3).

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:
   — 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:

        void f1(int); // #1
        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:

        void f(int (&&)[0]); // #1
        void f(double (&&)[0]); // #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

§ 12.4.3.2 334
— end example]

3.2  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 12.4.3.1.1, 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 (9.4.3) 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:

```
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&&)
```

— end example] or, if not that,

3.2.4  S1 and S2 include reference bindings (9.4.3) and S1 binds an lvalue reference to a function lvalue and S2 binds an rvalue reference to a function lvalue [Example:

```
struct A {
    & operator<< (int);
    void p() &;
    void p() &&;
};
&A operator<<(A&&, char);
A() << 1;  // calls A::operator<<(int)
A() << 'c';  // calls 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,

3.2.5  S1 and S2 differ only in their qualification conversion (7.3.5) and yield similar types T1 and T2, respectively, where T1 can be converted to T2 by a qualification conversion. [Example:

```
int f(void(&)());  // #1
int f(void(&&)());  // #2
void g();
int i1 = f(g);  // calls #1
```

— end example] or, if not that,

3.2.6  S1 and S2 include reference bindings (9.4.3), 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:

```
int f(const int &);
int f(int &);
```

§ 12.4.3.2
int g(const int &);
int g(int);

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:

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:

struct A {};
struct B : public A {};
struct C : public B {};
C* pc;
int f(A*);
int f(B*);
int i = f(pc);        // calls f(B*)
— end example]

(4.4.2) — 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,

(4.4.3) — conversion of A::* to B::* is better than conversion of A::* to C::*,

(4.4.4) — conversion of C to B is better than conversion of C to A,

(4.4.5) — conversion of B* to A* is better than conversion of C* to A*,

(4.4.6) — 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,
(4.4.7) — conversion of \( B::* \) to \( C::* \) is better than conversion of \( A::* \) to \( C::* \), and
(4.4.8) — conversion of \( B \) to \( A \) is better than conversion of \( C \) to \( A \).

[Note: 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 12.4.3); 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, 9.4.3, 9.4.4),
2. the left side of an assignment (7.6.19),
3. a parameter of a function (7.6.1.2),
4. a parameter of a user-defined operator (12.6),
5. the return value of a function, operator function, or conversion (8.7.3),
6. an explicit type conversion (7.6.1.3, 7.6.1.8, 7.6.3), or
7. a non-type template-parameter (13.4.2).

The function name can be preceded by the \& operator. [Note: 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 (7.3.13)) is identical to \( FT \). [Note: 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 (13.10.2.2), 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: As described in 13.10.1, if deduction fails and the function template name is followed by an explicit template argument list, the \textit{template-id} is then examined to see whether it identifies a single function template specialization. If it does, the \textit{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 7.6.2.1.

All functions with associated constraints that are not satisfied (13.5.2) 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 13.5.4. 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 13.7.6.2. After such eliminations, if any, there shall remain exactly one selected function.

[Example:

```c
int f(double);
int f(int);
int (*pfd)(double) = &f; // selects f(double)
int (*pfi)(int) = &f; // selects f(int)
int (*pfe)(...) = &f; // error: type mismatch
int (*krfi)(int) = f; // selects f(int)
int (*rfd)(double) = f; // selects f(double)
void g() {
  (int ())(*pfi) &f; // cast expression as selector
}
```
The initialization of \texttt{pfe} is ill-formed because no \texttt{f()} with type \texttt{int(...)} has been declared, and not because of any ambiguity. For another example,

```cpp
struct X {
    int f(int);
    static int f(long);
};
```

```cpp
int (X::*p1)(int) = &X::f;  // OK
int (*p2)(int) = &X::f;  // error: mismatch
int (*p3)(long) = &X::f;  // OK
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
```

—end example

7 [Note: If \texttt{f()} and \texttt{g()} are both overloaded functions, the cross product of possibilities must be considered to resolve \texttt{f(g)}, or the equivalent expression \texttt{f(g)}. — end note]

8 [Note: Even if \texttt{B} is a public base of \texttt{D}, we have

```cpp
D* f();
B* (*p1)() = &f;  // error
void g(D*);
void (*p2)(B*) = &g;  // error
```

—end note

12.6 Overloaded operators

A function declaration having one of the following \texttt{operator-function-ids} as its name declares an \texttt{operator function}. A function template declaration having one of the following \texttt{operator-function-ids} as its name declares an \texttt{operator function template}. A specialization of an operator function template is also an operator function. An operator function is said to \texttt{implement} the operator named in its \texttt{operator-function-id}.

\begin{verbatim}
operator-function-id:
    operator operator
    operator: one of
    new delete new[] delete[] co待ち () [] -> ->*
    - ! + - * / % ^ &
    | = += -= == /= %= ^= &= |=
    || <<= >>= <<= >> >= <= <= >= <=>
    |
\end{verbatim}

[Note: The operators \texttt{new[], delete[], ()}, and \texttt{[]} are formed from more than one token. The latter two operators are function call (7.6.1.2) and subscripting (7.6.1.1). — end note]

2 Both the unary and binary forms of

```cpp
+ - * &
```

can be overloaded.

3 [Note: The following operators cannot be overloaded:

```cpp
. * :: ?:
```

nor can the preprocessing symbols \# (15.6.2) and \## (15.6.3). — end note]

4 Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (12.6.1 – 12.6.6). They can be explicitly called, however, using the \texttt{operator-function-id} as the name of the function in the function call syntax (7.6.1.2). [Example:

```cpp
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

—end example]
The allocation and deallocation functions, \texttt{operator new}, \texttt{operator new[]}, \texttt{operator delete}, and \texttt{operator delete[]}, are described completely in 6.7.5.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 6.7.5.4.

The \texttt{co\_await} operator is described completely in 7.6.2.3. The attributes and restrictions found in the rest of this subclause do not apply to it unless explicitly stated in 7.6.2.3.

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 \texttt{=}, (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.

An operator function shall be a prefix unary, binary, function call, subscripting, class member access, increment, or decrement operator function.

[Note: The identities among certain predefined operators applied to basic types (for example, ++a \equiv 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]

An operator function cannot have default arguments (9.3.3.6), 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 this subclause.

Operators not mentioned explicitly in subclauses 12.6.2.1 through 12.6.6 act as ordinary unary and binary operators obeying the rules of 12.6.1 or 12.6.2.

### 12.6.1 Unary operators

A \textit{prefix unary operator function} is a function named \texttt{operator@} for a prefix unary-operator @ (7.6.2.1) that is either a non-static member function (11.4.1) with no parameters or a non-member function with one parameter. For a \texttt{unary-expression} of the form \@ \texttt{cast-expression}, the operator function is selected by overload resolution (12.4.1.2). If a member function is selected, the expression is interpreted as
\[
\text{cast-expression} \cdot \text{operator @} ()
\]
Otherwise, if a non-member function is selected, the expression is interpreted as
\[
\text{operator @} (\text{cast-expression})
\]
[Note: The operators ++ and -- (7.6.2.2) are described in 12.6.6. —end note]

The unary and binary forms of the same operator are considered to have the same name. [Note: Consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa. —end note]

### 12.6.2 Binary operators

A \textit{binary operator function} is a function named \texttt{operator@} for a binary operator @ that is either a non-static member function (11.4.1) with one parameter or a non-member function with two parameters. For an expression \texttt{x@y} with subexpressions \texttt{x} and \texttt{y}, the operator function is selected by overload resolution (12.4.1.2). If a member function is selected, the expression is interpreted as
\[
x \cdot \text{operator @} (y)
\]
Otherwise, if a non-member function is selected, the expression is interpreted as
\[
\text{operator @} (x, y)
\]

An \textit{equality operator function} is an operator function for an equality operator (7.6.10). A \textit{relational operator function} is an operator function for a relational operator (7.6.9). A \textit{three-way comparison operator function} is an operator function for the three-way comparison operator (7.6.8). A \textit{comparison operator function} is an equality operator function, a relational operator function, or a three-way comparison operator function.

### 12.6.2.1 Simple assignment

A \textit{simple assignment operator function} is a binary operator function named \texttt{operator=} . A simple assignment operator function shall be a non-static member function. [Note: Because only standard conversion sequences are considered when converting to the left operand of an assignment operation (12.4.3.1), an expression \texttt{x = y} with a subexpression \texttt{x} of class type is always interpreted as \texttt{x.operator=(y)}. —end note]
[Note: Since a copy assignment operator is implicitly declared for a class if not declared by the user (11.4.5), a base class assignment operator function is always hidden by the copy assignment operator function of the derived class. —end note]

[Note: 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. [Example:

```cpp
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]

12.6.3 Function call [over.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

```
postfix-expression ( expression-list_opt )
```

where the `postfix-expression` is of class type, the operator function is selected by overload resolution (12.4.1.1.2). If a surrogate call function for a conversion function named `operator conversion-type-id` is selected, the expression is interpreted as

```
postfix-expression . operator conversion-type-id () ( expression-list_opt )
```

Otherwise, the expression is interpreted as

```
postfix-expression . operator () ( expression-list_opt )
```

12.6.4 Subscripting [over.sub]

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

```
postfix-expression [ expr-or-braced-init-list ]
```

the operator function is selected by overload resolution (12.4.1.2). If a member function is selected, the expression is interpreted as

```
postfix-expression . operator [] ( expr-or-braced-init-list )
```

2 [Example:

```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
— end example]

§ 12.6.4
12.6.5 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

```
postfix-expression -> template_opt id-expression
```

the operator function is selected by overload resolution (12.4.1.2), and the expression is interpreted as

```
( postfix-expression . operator -> () ) -> template_opt id-expression
```

12.6.6 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. 

```cpp
Example:

```cpp
struct X {
    X& operator++(); // prefix ++a
    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++;
}
```

—end example

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 12.4.1.2 and are used for no other purpose. [Note: 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 12.4.1.2, 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 (7.3.6) (including e.g. `int` and `long` but excluding e.g. `char`). [Note: In all cases calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.]

---

126) Calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.
where a promoted integral type is required, an operand of unscoped enumeration type will be acceptable by way of the integral promotions. — end note]

3 In the remainder of this subclause, \( vq \) represents either \texttt{volatile} or no cv-qualifier.

4 For every pair \( (T, vq) \), where \( T \) is an arithmetic type other than \texttt{bool}, there exist candidate operator functions of the form

\[
vq \ T \& \text{operator}++(vq \ T\&); \\
T \text{operator}++(vq \ T\&, \text{int});
\]

5 For every pair \( (T, vq) \), where \( T \) is an arithmetic type other than \texttt{bool}, there exist candidate operator functions of the form

\[
vq \ T \& \text{operator}--(vq \ T\&); \\
T \text{operator}--(vq \ T\&, \text{int});
\]

6 For every pair \( (T, vq) \), where \( T \) is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form

\[
T* vq \ & \text{operator}++(T*vq\&); \\
T* vq \ & \text{operator}--(T*vq\&, \text{int}); \\
T* \text{operator}++(T*vq\&, \text{int}); \\
T* \text{operator}--(T*vq\&, \text{int});
\]

7 For every cv-qualified or cv-unqualified object type \( T \), there exist candidate operator functions of the form

\[
T \& \text{operator}*(T*);
\]

8 For every function type \( T \) that does not have cv-qualifiers or a \texttt{ref-qualifier}, there exist candidate operator functions of the form

\[
T \& \text{operator}*(T*);
\]

9 For every type \( T \) there exist candidate operator functions of the form

\[
T* \text{operator}+(T*);
\]

10 For every floating-point or promoted integral type \( T \), there exist candidate operator functions of the form

\[
T \text{operator}+- (T); \\
T \text{operator}-(T);
\]

11 For every promoted integral type \( T \), there exist candidate operator functions of the form

\[
T \text{operator}-(T);
\]

12 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

\[
cv12 \ T* \text{operator}>({cv1} \ C1*, \ \text{cv2} \ T \ C2::*);
\]

where \( cv12 \) 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

\[
LR \text{operator}*(L, R); \\
LR \text{operator}/(L, R); \\
LR \text{operator}+(L, R); \\
LR \text{operator}-(L, R); \\
bool \text{operator}==(L, R); \\
bool \text{operator}!=(L, R); \\
bool \text{operator}<=(L, R); \\
bool \text{operator}>(L, R); \\
bool \text{operator}<=(L, R); \\
bool \text{operator}>(L, R);
\]

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

\[
\texttt{std::strong\_ordering} \text{operator}<>(T, T);
\]

15 For every pair of floating-point types \( L \) and \( R \), there exists a candidate operator function of the form

\[
\texttt{std::partial\_ordering} \text{operator}<>(L, R);
\]
For every cv-qualified or cv-unqualified object type \( T \) there exist candidate operator functions of the form

\[
T* \quad \text{operator}+(\ast T, \text{std::ptrdiff_t}); \\
T* \quad \text{operator}[](\ast T, \text{std::ptrdiff_t}); \\
T* \quad \text{operator}-(\ast T, \text{std::ptrdiff_t}); \\
T* \quad \text{operator}+(\text{std::ptrdiff_t}, T*); \\
T* \quad \text{operator}[](\text{std::ptrdiff_t}, T*);
\]

For every \( T \), where \( T \) is a pointer to object type, there exist candidate operator functions of the form

\[
\text{std::ptrdiff_t} \quad \text{operator}-(\ast T, \ast T);
\]

For every \( T \), where \( T \) is an enumeration type or a pointer type, there exist candidate operator functions of the form

\[
\text{bool} \quad \text{operator}==(T, T); \\
\text{bool} \quad \text{operator}!=(T, T); \\
\text{bool} \quad \text{operator}< (T, T); \\
\text{bool} \quad \text{operator}> (T, T); \\
\text{bool} \quad \text{operator}<= (T, T); \\
\text{bool} \quad \text{operator}>= (T, T); \\
R \quad \text{operator}<>(T, T);
\]

where \( R \) is the result type specified in 7.6.8.

For every \( T \), where \( T \) is a pointer-to-member type or \( \text{std::nullptr_t} \), there exist candidate operator functions of the form

\[
\text{bool} \quad \text{operator}==(T, T); \\
\text{bool} \quad \text{operator}!=(T, T);
\]

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 \( R \).

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

\[
vq Lk \quad \text{operator}=(vq Lk, R); \\
vq Lk \quad \text{operator}==(vq Lk, R); \\
vq Lk \quad \text{operator}+=(vq Lk, R); \\
vq Lk \quad \text{operator}===(vq Lk, R); \\
vq Lk \quad \text{operator}-==(vq Lk, R);
\]

For every pair \((T, vq)\), where \( T \) is any type, there exist candidate operator functions of the form

\[
T* vq \& \quad \text{operator}=(T* vq \&, T*);
\]

For every pair \((T, vq)\), where \( T \) is an enumeration or pointer-to-member type, there exist candidate operator functions of the form

\[
vq Tk \quad \text{operator}=(vq Tk, T);
\]

For every pair \((T, vq)\), where \( T \) is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form

\[
T* vq \& \quad \text{operator}+=(T* vq \&, \text{std::ptrdiff_t}); \\
T* vq \& \quad \text{operator}===(T* vq \&, \text{std::ptrdiff_t});
\]

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

\[
vq Lk \quad \text{operator}%(vq Lk, R); \\
vq Lk \quad \text{operator}==(vq Lk, R); \\
vq Lk \quad \text{operator}<<=(vq Lk, R); \\
vq Lk \quad \text{operator}>>=(vq Lk, R); \\
vq Lk \quad \text{operator}^=(vq Lk, R); \\
vq Lk \quad \text{operator}|=(vq Lk, R);
\]
There also exist candidate operator functions of the form

```markdown
bool operator!(bool);
bool operator&&(bool, bool);
bool 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

```markdown
LR operator?:(bool, L, R);
```

where \( LR \) is the result of the usual arithmetic conversions (7.4) between types \( L \) and \( R \). [Note: 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

```markdown
T operator?:(bool, T, T);
```

### 12.8 User-defined literals

**[over.literal]**

**literal-operator-id:**

- `operator string-literal identifier`
- `operator user-defined-string-literal`

1. 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 16.5.4.3.5. A declaration whose `literal-operator-id` uses such a literal suffix identifier is ill-formed, no diagnostic required.

2. 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 (11.9.3)), 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`.

3. The declaration of a literal operator shall have a `parameter-declaration-clause` equivalent to one of the following:

```markdown
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 (9.3.3.6), 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 (13.7.3) 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: 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`.]

§ 12.8
they may have internal, module, or external linkage, they can be called explicitly, their addresses can be taken, etc. — end note

8 [Example:

```cpp
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
double operator""_Bq(long double); // OK: does not use the reserved identifier _Bq (5.10)
double operator""_Bq(long double); // 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 "C" void operator "" _m(long double); // error: C language linkage
```

— end example]
13 Templates

13.1 Preamble

1 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: The > token following the template-parameter-list of a template-declaration may be the product of replacing a >> token by two consecutive > tokens (13.3). —end note]

2 The declaration in a template-declaration (if any) shall

- declare or define a function, a class, or a variable, or
- 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
- define a member template of a class or class template, or
- be a deduction-guide, or
- be an alias-declaration.

3 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:]

```
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, -1i}};
};
```

—end example]
not be a template-id. [Note: That last component may 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 (13.7.5). — end note]

5 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.

6 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 (13.9.3), 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: 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]

7 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 13.7.5. Except that a function template can be overloaded either by non-template functions (9.3.3.5) with the same name or by other function templates with the same name (13.10.3), a template name declared in namespace scope or in class scope shall be unique in that scope.

8 An entity is templated if it is

(8.1) — a template,
(8.2) — an entity defined (6.2) or created (6.7.7) in a templated entity,
(8.3) — a member of a templated entity,
(8.4) — an enumerator for an enumeration that is a templated entity, or
(8.5) — the closure type of a lambda-expression (7.5.5.1) appearing in the declaration of a templated entity.

[Note: A local class, a local variable, or a friend function defined in a templated entity is a templated entity. — end note]

9 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 (13.5.2) 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: 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:

```cpp
template<int N> requires N == sizeof new unsigned short
int f();  // error: parentheses required around == expression
```

— end example] — end note]

10 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 (13.9.1) unless the corresponding specialization is explicitly instantiated (13.9.2) 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-specifier_opt concept-name
  nested-name-specifier_opt concept-name < template-argument-list_opt >

[Note: The > token following the template-parameter-list of a type-parameter may 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:

```cpp
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: A template argument may be a class template or alias template. For example,

```cpp
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 concept of type (13.7.8).

5 A type-parameter that starts with a type-constraint introduces the immediately-declared constraint of the type-constraint for the parameter. [Example:

```cpp
template<typename T> concept C1 = true;
template<typename... Ts> concept C2 = true;
template<typename T, typename U> concept C3 = true;

template<C1 T> struct s1; // associates C1<T>
template<C1... T> struct s2; // associates (C1<T> & & ⋯)
template<C2... T> struct s3; // associates (C2<T> & & ⋯)
template<C3<int> T> struct s4; // associates C3<T, int>
template<C3<int>... T> struct s5; // associates (C3<T, int> & & ⋯)
```

—end example]

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 (9.2.8.5), or

127) 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.
A structural type is one of the following:

- a scalar type, or
- an lvalue reference type, or
- a literal class type with the following properties:
  - all base classes and non-static data members are public and non-mutable and
  - the types of all bases classes and non-static data members are structural types or (possibly multi-dimensional) array thereof.

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. [Note: 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 (7.5.4.1).] [Example:

```cpp
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
}
```

[Note: A non-type template-parameter cannot be declared to have type cv void.] [Example:

```cpp
template<void> class X; // error
template<void* pv> class Y; // OK
```

[End note]

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:

```cpp
template<int* a> struct R { /* ... */ };  
template<int b[5]> struct S { /* ... */ };  
int p;
R<&p> w; // OK
S<&p> x; // OK due to parameter adjustment
int v[5];
R<int& y; // OK due to implicit argument conversion
S<int& z; // OK due to both adjustment and conversion
```

[End example]

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 (9.3.3.5).

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 (13.7.3). 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 outside of the member’s class. A default template-argument shall 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 (9.3.3.6). [Example:

```cpp
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
```

is equivalent to

```cpp
template<class T1 = int, class T2 = int> class A;
```

—end example]

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 (9.3.3.5) of the function template or has a default argument (13.10.2). A template parameter of a deduction guide template (13.7.1.2) that does not have a default argument shall be deducible from the parameter-type-list of the deduction guide template. [Example:

```cpp
template<class T1 = int, class T2> class B; // error
```

// U can be neither deduced from the parameter-type-list nor specified
```cpp
template<class... T, class... U> void f() { } // error
template<class... T, class U> void g() { } // error
```

—end example]

A template-parameter shall not be given default arguments by two different declarations in the same scope. [Example:

```cpp
template<class T = int> class X;
template<class T = int> class X { /* ... */ }; // error
```

—end example]

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. [Example:

```cpp
template<int i = 3 > 4>
```

class X { /* ... */ }; // syntax error
```cpp
template<int i = (3 > 4) >
```

class Y { /* ... */ }; // OK

—end example]

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:

```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 (9.3.3.5), then the template-parameter is a template parameter pack (13.7.3). 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:

```cpp
template <class... Types>
    class Tuple; // Types is a template type parameter pack

template <class T, int... Dims>
    struct multi_array; // Dims is a non-type template parameter pack

template <class... T>
    struct value_holder {
        template <T... Values> struct apply { }; // Values is a non-type template parameter pack
    };

template <class... T, T... Values>
    struct static_array; // error: Values expands template type parameter pack within the same template parameter list

    // 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:

```
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...
    template-argument-list , template-argument...

template-argument:
    constant-expression
    type-id
    id-expression
```

[Note: The name lookup rules (6.5) are used to associate the use of a name with a template declaration; that is, to identify a name as a template-name. —end note]

For a template-name to be explicitly qualified by the template arguments, the name must be considered to refer to a template. [Note: Whether a name actually refers to a template cannot be known in some cases until after argument dependent lookup is done (6.5.2). —end note] A name is considered to refer to a template if name lookup finds a template-name or an overload set that contains a function template. A name is also considered to refer to a template if it is an unqualified-id followed by a < and name lookup either finds one or more functions or finds nothing.

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 >128 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: The second > token produced by this replacement rule may terminate an enclosing template-id construct or it may be part of a different construct (e.g., a cast). —end note] [Example:

```
template<int i> class X { /* ... */ };

X< 1>2 > x1; // syntax error
X<(1>2)> x2; // OK
```

128) 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> 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 (9.2.8.3), 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 (13.8.2.1), the member template name shall be prefixed by the keyword `template`. [Example:

```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: The keyword `template` may not be applied to non-template members of class templates. —end note] [Note: 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:

```cpp
template <class T> struct A {
    void f(int);
    template <class U> void f(U);
};

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 T, template <class X> class TT = T::template C> struct D { }
D<B<int>> db;
—end example]

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 (13.7.3),  
(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
— 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 (13.10.2). [Example:

```c
template<class T, T::type n = 0> class X;
struct S {
    using type = int;
};
using T1 = X<S, int, int>;<comment error: too many arguments
using T2 = X<>; <comment error: no default argument for first template parameter
using T3 = X<1>; <comment error: value 1 does not match type-parameter
using T4 = X<int>; <comment error: substitution failure for second template parameter
using T5 = X<S>; <comment OK
```]

When the template-name of a simple-template-id names a constrained non-function template or a constrained template 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 (13.8.2.4), the associated constraints (13.5.2) of the constrained template shall be satisfied (13.5.1). [Example:

```c
template<typename T> concept C1 = sizeof(T) != sizeof(int);

template<C1 T> struct S1 { };
template<C1 T> using Ptr = T*;
S1<int>* p; // error: constraints not satisfied
Ptr<int> p; // error: constraints not satisfied
```]

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 (13.5.2) is satisfied (13.5.1) by the specified template arguments and false otherwise. [Note: 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:

```c
template<typename T> concept C2 = sizeof(T) == 1;
template struct S<char[2]>; // error: constraints not satisfied
```]

13.4 Template arguments [temp.arg]

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 (13.7.3), it will correspond to zero or more template-arguments. [Example:

```cpp
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);  // T& operator[] is a non-explicit member function
  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]

2 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. [Example:

```cpp
template<class T> void f();
template<int I> void f();

void g() {
  f<int>()();  // int() is a type-id: call the first f()
}

—end example]

3 The name of a template-argument shall be accessible at the point where it is used as a template-argument. [Note: 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:

```cpp
template<class T> class X {
  static T t;
};

class Y {
private:
  struct S { /* ... */ };  // OK: S is accessible
  // X<S>::S> has a static member of type Y::S
  X<Y::S> x;  // OK: even though Y::S is private
};

X<Y::S> y;  // error: S not accessible

—end example]

—end example]

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:

```cpp
template <template <class TT> class T> class A {
  typename T<int>::S s;
};
```

}]
template <class U> class B {
private:
   struct S { /* ... */ }
};

A<B> b; // error: A has no access to B::S
—end example

4 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:

```c
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
—end example
```

5 An explicit destructor call (11.4.6) for an object that has a type that is a class template specialization may explicitly specify the template-arguments. [Example:

```c
template<class T> struct A {
   ~A();
};
void f(A<int>** p, A<int>** q) {
   p->A<int>::~A(); // OK: destructor call
   q->A<int>::~A<int>(); // OK: destructor call
}
—end example
```

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-arguments do not match the template-parameters 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 (13.9.1) when the value of that default argument is needed. [Example:

```c
template<typename T, typename U = int> struct S { }
S<bool>** p; // the type of p is S<bool, int>*
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 (13.7.3).

### 13.4.1 Template type arguments

1 A template-argument for a template-parameter which is a type shall be a type-id.

2 [Example:

```c
template <class T> class I { }
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: A template type argument may be an incomplete type (6.8). — end note]

13.4.2 Template non-type arguments  [temp.arg.nontype]

If the type \( T \) of a template-parameter (13.2) contains a placeholder type (9.2.8.5) or a placeholder for a deduced class type (9.2.8.6), the type of the parameter is the type deduced for the variable \( x \) in the invented declaration

\[
T \ x = \text{template-argument} \;
\]

If a deduced parameter type is not permitted for a template-parameter declaration (13.2), the program is ill-formed.

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: 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]

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):

1. a temporary object (6.7.7),
2. a string literal object (5.13.5),
3. the result of a typeid expression (7.6.1.7),
4. a predefined __func__ variable (9.5.1), or
5. a subobject (6.7.2) of one of the above.

[Example:

\[
\text{template<const int* pci> struct X { /* ... */ };}
\]

\[
\text{int ai[10];}
\]

\[
\text{X<ai> xi; \quad \text{// array to pointer and qualification conversions}}
\]

\[
\text{struct Y { /* ... */ };}
\]

\[
\text{template<const Y& b> struct Z { /* ... */ };}
\]

\[
\text{Y y;}
\]

\[
\text{Z<y> z; \quad \text{// no conversion, but note extra cv-qualification}}
\]

\[
\text{template<int (&pa)[5]> struct W { /* ... */ };}
\]

\[
\text{int b[5];}
\]

\[
\text{W<b> w; \quad \text{// no conversion}}
\]

\[
\text{void f(char);}
\]

\[
\text{void f(int);}
\]

\[
\text{template<\void (pf)(int)> struct A { /* ... */ };}
\]

\[
\text{A<&f> a; \quad \text{// selects f(int)}}
\]

\[
\text{template<\auto n> struct B { /* ... */ };}
\]

\[
\text{B<5> b1; \quad \text{// OK, template parameter type is int}}
\]

\[
\text{B<\char> b2; \quad \text{// OK, template parameter type is char}}
\]

\[
\text{B<2.5> b3; \quad \text{// OK, template parameter type is double}}
\]

\[
\text{B<\void(0)> b4; \quad \text{// error: template parameter type cannot be void}}
\]

— end example]

5 [Note: A string-literal (5.13.5) is not an acceptable template-argument for a template-parameter of non-class type. [Example:

\[
\text{template<class T, T p> class X { /* ... */}}
\]

]}
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: A temporary object is not an acceptable template-argument when the corresponding template-parameter has reference type. [Example:

template<const int& CRI> struct B { /* ... */ };
B<1> b1;
// error: temporary would be required for template argument
int c = 1;
B<c> 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]

13.4.3 Template template arguments
[temp.arg.template] 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.

Any partial specializations (13.7.5) 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:

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 {  // error: subobject of temporary object used to initialize
    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]

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 (13.7.6.1), and for template template-parameters, each of their corresponding template-parameters matches, recursively. When P’s template-head contains a template parameter pack (13.7.3), the template parameter pack will

§ 13.4.3 357
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:**

```cpp
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 P> class X> 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
Y<C> yc; // OK
Z<D> zd; // OK
```

— end example]

**Example:**

```cpp
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<17>> eC; // error: C does not match TT in partial specialization
eval<D<int, 17>> eD; // error: D does not match TT in partial specialization
eval<E<int, float>> eE; // error: E does not match TT in partial specialization
```

— end example]

**Example:**

```cpp
template<typename T> concept C = requires (T t) { t.f(); };
template<typename T> concept D = C<T> && requires (T t) { t.g(); };

template<template<C> class P> struct S { };
template<template<C> class P> struct X { };
template<template<C> class P> struct Y { };
template<template<C> class P> 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 (13.7.6.2). Given an invented class template `X` with the `template-head` of `A` (including default arguments and `requires-clause`, if any):

(4.1) Each of the two function templates has the same template parameters and `requires-clause` (if any), respectively, as `P` or `A`.

§ 13.4.3
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 pack expansion \( PP \ldots \) (13.7.3); 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

[Note: This subclause defines the meaning of constraints on template arguments. The abstract syntax and satisfaction rules are defined in 13.5.1. Constraints are associated with declarations in 13.5.2. Declarations are partially ordered by their associated constraints (13.5.4). — end note]

#### 13.5.1 Constraints

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** (13.5.1.1),
2. **Disjunctions** (13.5.1.1), and
3. **Atomic constraints** (13.5.1.2).

In order for a constrained template to be instantiated (13.9), its associated constraints (13.5.2) shall be satisfied as described in the following subclauses. [Note: 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 (12.4.2) requires the satisfaction of constraints on functions and function templates. — end note]

#### 13.5.1.1 Logical operations

There are two binary logical operations on constraints: conjunction and disjunction. [Note: These logical operations have no corresponding C++ syntax. For the purpose of exposition, conjunction is spelled using the symbol \( \land \) and disjunction is spelled using the symbol \( \lor \). The operands of these operations are called the left and right operands. In the constraint \( A \land B \), \( A \) 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 not satisfied, the disjunction is not satisfied. Otherwise, the disjunction is satisfied if and only if the second operand is satisfied.

**Example:**

```cpp
template<typename T>
constexpr 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 \land get_value<T>()

void f(int); // OK: calls f(int)
```

In the satisfaction of the associated constraints (13.5.2) of \( f \), the constraint \( \text{sizeof(char)} > 1 \) is not satisfied; the second operand is not checked for satisfaction. — end example]

A logical negation expression (7.6.2.1) 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 13.7.6.1 do not subsume one another under 13.5.4. Furthermore, if substitution to determine whether an atomic constraint is satisfied (13.5.1.2) encounters a substitution failure, the constraint is not satisfied, regardless of the presence of a negation operator. [Example:

```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 (13.5.1.2)
// 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

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>;
template <class T> int f4(T) requires (!sad_nested_type<T>);
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 —end note

13.5.1.2 Atomic constraints [temp.constr.atomic]

1 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 (13.5.2). [Note: Atomic constraints are formed by constraint normalization (13.5.3). 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 (13.7.6.1) 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: The comparison of parameter mappings of atomic constraints operates in a manner similar to that of declaration matching with alias template substitution (13.7.7). Example:

```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 M> void f()
requires Add1<2 * M>;
template <unsigned M> int f()
requires AddOne<2 * M> && true;
int x = f<0>(); // OK, the atomic constraints from concept C in both fs 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 M> void g(Add1Ty<2 * M> *);
template <unsigned M> void g(AddOneTy<2 * M> *);

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:

```cpp
template <unsigned N> void f2()
requires Add1<2 * N>;
```
template <unsigned N> int f2()
    requires Add1<N * 2> && true;
void h2() {
    f2<0>(); // ill-formed, no diagnostic required:
    // requires determination of subsumption between atomic constraints that are
    // functionally equivalent but not equivalent
}

— end example] — end note

3 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 (7.3.1) is performed if necessary, and E shall be a constant expression of type bool. The constraint is satisfied if and only if evaluation of E results in 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:

```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
}

— end example]

13.5.2 Constrained declarations [temp.constr.decl]

1 A template declaration (13.1) or templated function declaration (9.3.3.5) 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

2 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.

3 A declaration's associated constraints are defined as follows:

(3.1) — If there are no introduced constraint-expressions, the declaration has no associated constraints.

(3.2) — Otherwise, if there is a single introduced constraint-expression, the associated constraints are the normal form (13.5.3) of that expression.

(3.3) — Otherwise, the associated constraints are the normal form of a logical AND expression (7.6.14) whose operands are in the following order:

(3.3.1) — the constraint-expression introduced by each type-constraint (13.2) in the declaration’s template-parameter-list, in order of appearance, and

(3.3.2) — the constraint-expression introduced by a requires-clause following a template-parameter-list (13.1), and

(3.3.3) — the constraint-expression introduced by each type-constraint in the parameter-type-list of a function declaration, and

(3.3.4) — the constraint-expression introduced by a trailing requires-clause (9.3) of a function declaration (9.3.3.5).

The formation of the associated constraints establishes the order in which constraints are instantiated when checking for satisfaction (13.5.1). [Example:
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> void f4(T) requires C2<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

When determining whether a given introduced constraint-expression C₁ of a declaration in an instantiated specialization of a templated class is equivalent (13.7.6.1) 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: This can happen when determining which member template is specialized by an explicit specialization declaration. —end note] [Example:

```
template <class T> concept C = true;
template <class T> struct A {
    template <class U> U f(U) requires C<typename T::type>;
    template <class U> U f(U) requires C<T>;
};
```

```
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]

### 13.5.3 Constraint normalization [temp.constr.normal]

1 The normal form of an expression E is a constraint (13.5.1) 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 (13.5.1.1) 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:

```
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.

2 The process of obtaining the normal form of a constraint-expression is called normalization. [Note: Normalization of constraint-expressions is performed when determining the associated constraints (13.5.1) of a declaration and when evaluating the value of an id-expression that names a concept specialization (7.5.4). —end note]
A constraint \( P \) subsumes a constraint \( Q \) if and only if, for every disjunctive clause \( P_i \) in the disjunctive normal form of \( P \), \( P_i \) subsumes every conjunctive clause \( Q_j \) in the conjunctive normal form of \( Q \), where

1. A disjunctive clause \( P_i \) subsumes a conjunctive clause \( Q_j \) if and only if there exists an atomic constraint \( P_{ia} \) in \( P_i \) for which there exists an atomic constraint \( Q_{jb} \) in \( Q_j \) such that \( P_{ia} \) subsumes \( Q_{jb} \), and
2. An atomic constraint \( A \) subsumes another atomic constraint \( B \) if and only if there exists an atomic constraint \( B_i \) in \( B \) such that \( A \) subsumes \( B_i \).

Example: Let \( A \) and \( B \) be atomic constraints (13.5.1.2). The constraint \( A \land B \) subsumes \( A \), but \( A \) does not subsume \( A \land B \). The constraint \( A \) subsumes \( A \lor B \), but \( A \lor B \) does not subsume \( A \). Also note that every constraint subsumes itself. —end example

[Note: The subsumption relation defines a partial ordering on constraints. This partial ordering is used to determine

1. The best viable candidate of non-template functions (12.4.3),
2. The address of a non-template function (12.5),
3. The matching of template template arguments (13.4.3),
4. The partial ordering of class template specializations (13.7.5.2), and
5. The partial ordering of function templates (13.7.6.2).

—end note]

A declaration \( D_1 \) is at least as constrained as a declaration \( D_2 \) if

1. \( D_1 \) and \( D_2 \) are both constrained declarations and \( D_1 \)'s associated constraints subsume those of \( D_2 \); or
2. \( D_2 \) has no associated constraints.

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:

\[
\begin{align*}
\text{template<typename T> concept C1 = requires(T t) { --t; };}\\
\text{template<typename T> concept C2 = C1<T> \&\& \text{requires(T t) { *t; };};}\\
\text{template<C1 T> void f(T);} & \quad \text{// #1}\\
\text{template<C2 T> void f(T);} & \quad \text{// #2}\\
\text{template<typename T> void g(T);} & \quad \text{// #3}\\
\text{template<C1 T> void g(T);} & \quad \text{// #4}\\
\end{align*}
\]

\[
\begin{align*}
f(0); & \quad \text{// selects #1}\\
f((\text{int}*)0); & \quad \text{// selects #2}\\
g(\text{true}); & \quad \text{// selects #3 because C1<bool> is not satisfied}\\
g(0); & \quad \text{// selects #4}
\end{align*}
\]

Example: 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 conjunctive clauses are \( (A \land B) \) and \( (A \land C) \). —end example]

131) A constraint is in conjunctive normal form when it is a conjunction of clauses where each clause is a conjunction of atomic constraints. [Example: For atomic constraints \( A \), \( B \), and \( C \), the constraint \( A \land (B \lor C) \) is in conjunctive normal form. Its conjunctive clauses are \( A \land (B \lor C) \). —end example]
13.6 Type equivalence

Two template-ids are the same if

1. their template-names, operator-function-ids, or literal-operator-ids refer to the same template, and
2. their corresponding type template-arguments are the same type, and
3. their corresponding non-type template-arguments are template-argument-equivalent (see below) after conversion to the type of the template-parameter, and
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

1. they are of integral type and their values are the same, or
2. they are of floating-point type and their values are identical, or
3. they are of type std::nullptr_t, or
4. they are of enumeration type and their values are the same, or
5. they are of pointer type and they have the same pointer value, or
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
7. they are of reference type and they refer to the same object or function, or
8. they are of array type and their corresponding elements are template-argument-equivalent, or
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
10. they are of class type and their corresponding direct subobjects and reference members are template-argument-equivalent.

[Example:

```c
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

```c
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.

```c
template<class T> struct X { };  
template<class> struct Y { };  
template<class T> using Z = Y<T>;  
X<Y<int> > y;  
X<Z<int> > z;  
```
declares y and z to be of the same type. —end example]

If an expression e is type-dependent (13.8.2.2), decltype(e) denotes a unique dependent type. Two such decltype-specifiers refer to the same type only if their expressions are equivalent (13.7.6.1). [Note: However, such a type may be aliased, e.g., by a typedef-name. — end note]

13.7 Template declarations

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:

132) The identity of enumerators is not preserved.
133) An array as a template-parameter decays to a pointer.
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: However, this syntax is allowed in class template partial specializations (13.7.5).
— end note]

2 For purposes of name lookup and instantiation, default arguments, type-constraints, requires-clause (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 (8.5.1) are considered definitions.

3 Because an alias-declaration cannot declare a template-id, it is not possible to partially or explicitly specialize
an alias template.

13.7.1 Class templates  [temp.class]

1 A class template defines the layout and operations for an unbounded set of related types.

2 [Example: 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:

    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]

3 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 (13.7.6.1).
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:

    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();
    }

    template<class ... Types> void B<Types ...>::f3() { } // OK
    template<class ... Types> void B<Types>::f4() { } // error

    template<typename T> concept C = true;
    template<typename T> concept D = true;

    template< typename T> struct S {
    void f();
    void g();

    § 13.7.1 365
void h();
template<U> struct Inner;

};

template<typename T> void S<T>::g() { } // error: no matching declaration for S<T>

template<typename T> requires C<T>
void S<T>::h() { } // functionally equivalent but not equivalent

— end example

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 (9.2.8.3).

13.7.1.1 Member functions of class templates [temp.mem.func]

A member function of a class template may be defined outside of the class template definition in which it is declared. [Example:

template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[] (int i); // subscript function might be defined like this:
    T& elem(int i) { return v[i]; }
};

declares three member functions of a class template. The subscript function might be defined like this:

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:

template<typename T> concept C = requires {
    T::type;
};

template<typename T> struct S {
    void f() requires C<T>;
    void g() requires C<T>;
};

void S<T>::f() requires C<T> { } // OK

— end example]

The template-arguments for a member function of a class template are determined by the template-arguments of the type of the object for which the member function is called. [Example: The template-argument for Array<T>::operator[] will be determined by the Array to which the subscripting operation is applied.

Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7; // Array<int>::operator[]
v2[3] = dcomplex(7,8); // Array<dcomplex>::operator[]
— end example]
13.7.1.2 Deduction guides

Deduction guides are used when a template-name appears as a type specifier for a deduced class type (9.2.8.6). Deduction guides are not found by name lookup. Instead, when performing class template argument deduction (12.4.1.8), any deduction guides declared for the class template are considered.

deduction-guide:
    explicit-specifieropt template-name ( parameter-declaration-clause ) -> simple-template-id ;

[Example:
    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]

The same restrictions apply to the parameter-declaration-clause of a deduction guide as in a function declaration (9.3.3.5). The simple-template-id shall name a class template specialization. The template-name shall be the same identifier as the template-name of the simple-template-id. A deduction-guide shall be declared in the same scope as the corresponding class template and, for a member class template, with the same access. Two deduction guide declarations in the same translation unit for the same class template shall not have equivalent parameter-declaration-clauses.

13.7.1.3 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: The member class must be defined before its first use that requires an instantiation (13.9.1). For example,
    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 { };
    A<int>::B b2; // OK: requires A::B to be defined
—end note]

13.7.1.4 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:
    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:
13.7.1.5 Enumeration members of class templates

An enumeration member of a class template may be defined outside the class template definition. [Example:

```cpp
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]

13.7.2 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 (13.7.6.1). [Example:

```cpp
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:

```cpp
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>
```
—end example]

2 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 (9.3.3.5) 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:

```cpp
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) { } // member function template specialization
```
```c
int main() {
    A<char> ac;       // non-template
    ac.f(1);         // non-template
    ac.f('c');       // template
    ac.f<>()(1);     // template
}
— end example

A member function template shall not be virtual. [Example:

```c
template <class T> struct AA {
    template <class C> virtual void g(C);    // error
    virtual void f();                        // OK
};
— end example

A specialization of a member function template does not override a virtual function from a base class. [Example:

```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<>()(i); }   // overriding function that calls the template instantiation
};
— end example

A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type. [Example:

```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*()
}
— end example] [Note: There is no syntax to form a template-id (13.3) by providing an explicit template argument list (13.10.1) for a conversion function template (11.4.7.2). — 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 (13.10.2.3), 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 (12.4.3.2) and partial ordering (13.7.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

13.7.3 Variadic templates [temp.variadic]

A template parameter pack is a template parameter that accepts zero or more template arguments. [Example:

```c
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
```
2 A function parameter pack is a function parameter that accepts zero or more function arguments. [Example:

```cpp
template<class ... Types> void f(Types ... args);
```

```cpp
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]

3 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:

```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]

4 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:

- In a function parameter pack (9.3.3.5); the pattern is the parameter-declaration without the ellipsis.
- In a using-declaration (9.9); the pattern is a using-declarator.
- In a template parameter pack that is a pack expansion (13.2):
  - 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 (11.10.2) 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 (7.5.5.2); the pattern is the capture without the ellipsis.
- In a sizeof... expression (7.6.2.4); the pattern is an identifier.
- In a fold-expression (7.5.6); the pattern is the cast-expression that contains an unexpanded pack.

[Example:

```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]
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.

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:

```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

template<class ... Args>
void g(Args ... args) {
    f(const_cast<const Args*>(&args)...); // OK: Args is expanded by the function parameter pack args
    f(5 ...); // error: pattern does not contain any packs
    f(args); // error: pack "args" is not expanded
    f(h(args ...) + args ...); // OK: first "args" expanded within h,
                                    // second "args" expanded within f
}

— end example]

The instantiation of a pack expansion that is neither a sizeof... expression nor a fold-expression produces a list of elements $E_1, E_2, \cdots, E_N$, 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:

1. 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;
2. 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
3. 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: 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:

```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

— end example]

The instantiation of a sizeof... expression (7.6.2.4) produces an integral constant containing the number of elements in the pack it expands.

The instantiation of a fold-expression produces:
In each case, \( op \) is the fold-operator, \( N \) is the number of elements in the pack expansion parameters, and each \( E_i \) is generated by instantiating the pattern and replacing each pack expansion parameter with its \( i \)th element. For a binary fold-expression, \( E \) is generated by instantiating the cast-expression that did not contain an unexpanded pack. [Example:

```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 17: Value of folding empty sequences**

<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>

### 13.7.4 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 (13.10.2.6), otherwise,
4. the name shall be an unqualified-id that declares (or redeclares) a non-template function.

[Example:

```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 F> 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:

```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 (13.9.1), partially specialized (13.7.5) or explicitly specialized (13.9.3), are friends of the class containing the template friend declaration. [Example:

```cpp
class X {
    template<class T> friend struct A;
    class Y { }; // OK
};
template<class T> struct A { X::Y ab; };  // OK
template<class T> struct A<T*> { X::Y ab; }; // OK
—end example]

4 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 (13.10.2.5). 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:

```cpp
template<class T> struct A {
    struct B { }; // grants friendship to A<int>::B even though
    void f();
    struct D {
        void g();
    };       // is not a specialization of A<T>::B
    T h();
    template<T> T i();
};
template<> struct A<int> {
    struct B { }; // does not grant friendship to A<int>::f()
    int f();
    struct D {
        void g();
    };       // because its return type does not match
    template<int> 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
    template<class T> friend void A<T>::f();    // it is not a specialization of A<T>::B
    template<class T> friend void A<T>::D::g(); // does not grant friendship to A<int>::f()
    template<class T> friend int *A<T*>::h();   // error: A<T>::D does not end with a simple-template-id
    template<class T> friend int A<T*>::h();    // grants friendship to A<int>::h() and A<float*>::h()

§ 13.7.4 373

373
template<class T> template<T U> 
    // grants friendship to instantiations of A<T>::i() and
    friend T A<T>::i();    // to A<int>::i(), and thereby to all specializations
    // of those function templates

— end example |

5 [Note: A friend declaration may first declare a member of an enclosing namespace scope (13.8.5). — end note]

6 A friend template shall not be declared in a local class.

7 Friend declarations shall not declare partial specializations. [Example:

    template<class T> class A { };
    class X {
    template<class T> friend class A<T*>;     // error
    };

    — end example]

8 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.

9 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.

13.7.5 Class template partial specializations [temp.class.spec]

1 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 (13.7.5.1). 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 (13.7.5.3).

3 [Example:

    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:

    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 (13.5.4). — 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: The template argument list for the primary template in the example above is \(<T_1, T_2, I>\). —end example] [Note: 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 (9.8.1.2, 11.4, 13.7.2). [Example:

```cpp
template<class T> struct A {
    struct C {
        template<class T2> struct B { };  // partial specialization #1
        template<class T2> struct B<T2**> { };  // partial specialization #2
    }
};
// partial specialization of A<T>::C::B<T2>

template<class T> template<class T2>
struct A<T>::C::B<T2*> { };  // #2

A<short>::C::B<int*> absip;  // uses partial specialization #2
—end example]
```

7 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:

```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
—end example]
```

8 A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

9 Within the argument list of a class template partial specialization, the following restrictions apply:

(9.1) — The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization. [Example:

```cpp
template <class T, T t> struct C {};
template <class T> struct C<T, 1>;  // error

template< int X, int (*array_ptr)[X] > class A {}; int array[5];
template< int X > class A<X,&array> { };  // error
—end example]
```

(9.2) — The specialization shall be more specialized than the primary template (13.7.5.2).

(9.3) — The template parameter list of a specialization shall not contain default template argument values.\(^{134}\)

(9.4) — An argument shall not contain an unexpanded pack. If an argument is a pack expansion (13.7.3), it shall be the last argument in the template argument list.

\(^{134}\) There is no way in which they could be used.
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: The template arguments may 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]

13.7.5.1 Matching of class template partial specializations

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 (13.7.5.2) 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.

3. 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 (13.10.2), and the deduced template arguments satisfy the associated constraints of the partial specialization, if any (13.5.2). [Example:

```
template<class T1, class T2, int I> class A { }; // #1
template<class T, int I> class A<T, T*, I> { }; // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { }; // #3
template<class T> class A<int, T*, 5> { }; // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { }; // #5
```

A<int, int, I> a1; // uses #1
A<int, int*, I> a2; // uses #2, T is int, I is 1
A<int, char*, 5> a3; // uses #4, T is char
A<int, char*, I> a4; // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5; // ambiguous: matches #3 and #5

— end example] [Example:

```
template<typename T> concept C = requires (T t) { t.f(); }; 
```

```
template<typename T> struct S { }; // #1
template<C T> struct S<T> { }; // #2
```

```
struct Arg { void f(); }; // uses #1; the constraints of #2 are not satisfied
```

```
S<int> s1; // uses #1 and #2; both constraints are satisfied but #2 is more specialized
```

— end example]

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:

```
template <int I, int J> struct A {};  
```

```
template <int I> struct A<I, I> {};  // OK
```

```
template <int I, int J, int K> struct B {};  
```

```
template <int I> struct B<I, I*2, 2> {};  // OK
```

— end example]

In a type name that refers to a class template specialization, (e.g., A<int, int, I>) 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.
13.7.5.2 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 (13.7.6.2):

1. Each of the two function templates has the same template parameters and associated constraints (13.5.2) 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:

```c++
template<int I, int J, class T> class X { };
template<int I, int J> class X<I, int> { };  // #1
template<int I> class X<I, I> { };  // #2

template<int I0, int J0> void f(X<I0, J0, int>);
// A

template<int I0> void f(X<I0, int, int>);
// B

template<auto v> class Y { };

template<auto* p> class Y<p> { };  // #3

template<auto** pp> class Y<pp> { };  // #4

template<auto* p0> void g(Y<p0>);  // C

template<auto** pp0> void g(Y<pp0>);  // D
```

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:

```c++
// primary class template
template<class T, int I> struct A {  
    void f();
};

// member of primary class template
template<class T, int I> void A<T,I>::f() { }
```}

13.7.5.3 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:

```c++
// primary class template
template<class T, int I> struct A {
    void f();
};

// member of primary class template
template<class T, int I> void A<T,I>::f() { }
```}
// class template partial specialization
template<class T> struct A<T,2> {
    void f();
    void g();
    void h();
};

// member of class template partial specialization
template<class T> void A<T,2>::g() { }

// explicit specialization
template<> void A<char,2>::h() { }

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 (13.9.1, 13.9.2), 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:

```cpp
template<class T> struct A {
    template<class T2> struct B {};  // #1
    template<class T2> struct B<T2*> {}; // #2
};
template<> template<class T2> struct A<short>::B {}; // #3
A<char>::B<int*> abcip;  // uses #2
A<short>::B<int*> absip;  // uses #3
A<char>::B<int> abci;    // uses #1
—end example
```

13.7.6 Function templates [temp.fct]

A function template defines an unbounded set of related functions. [Example: 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
```

A function template can be overloaded with other function templates and with non-template functions (9.3.3.5). 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.\(^{135}\)

13.7.6.1 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:

\[^{135}\] 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.

1
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.

Template specializations are not necessarily distinct, even if overload resolution alone cannot distinguish them.

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.

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 they name the same entities 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.

Lambda-expressions are never considered equivalent. For determining whether two dependent names 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.
template <class T> void spam(decltype([]() {}) (*s)[sizeof(T)]);

— end example —

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: 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:

- they declare template parameters of the same kind,
- if either declares a template parameter pack, they both do,
- if they declare non-type template parameters, they have equivalent types ignoring the use of type-constraints for placeholder types, and
- 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 (13.5.1) 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: 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:

```cpp
// guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
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 —
```

### 13.7.6.2 Partial ordering of function templates

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (13.10.2) 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:

- during overload resolution for a call to a function template specialization (12.4.3);
- when the address of a function template specialization is taken;
- when a placement operator delete that is a function template specialization is selected to match a placement operator new (6.7.5.4.2, 7.6.2.7);
1.4  when a friend function declaration (13.7.4), an explicit instantiation (13.9.2) or an explicit specialization (13.9.3) refers to a function template specialization.

2  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.

3  To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (13.7.3) 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: 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 (12.4.1.2) 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 $cv$ that is a member of a class $A$:

3.1  The type $X(M)$ is “rvalue reference to $cv A$” if the optional ref-qualifier of $M$ is && or if $M$ has no 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.

3.2  Otherwise, $X(M)$ is “lvalue reference to $cv A$”.

[Note: 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:

```c++
struct A {};
template<class T> struct B {
    template<class R> int operator*(R&);
};

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]

4  Using the transformed function template’s function type, perform type deduction against the other template as described in 13.10.2.4.

[Example:

```c++
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*)
}
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: 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:

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:

template<class T, class U> struct A { };

template<class T, class U> void f(U, A<U, T>* p = 0);  // #1
template<class T, class U> void f(U, A<U, 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:

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 (9.3.3.5)) 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:

(6.2.1.1) — If, for either template, some of the template parameters are not deducible from their function
parameters, neither template is more specialized than the other.

(6.2.1.2) — If there is either no reordering or more than one reordering of the associated template-parameter-
list such that

§ 13.7.6.2
the corresponding template-parameters of the template-parameter-lists are equivalent and
neither template is more specialized than the other.

Otherwise, if the corresponding template-parameters of the template-parameter-lists are not equivalent (13.7.6.1) 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 one template is more constrained than the other (13.5.4), the more constrained template is more specialized than the other.

Otherwise, neither template is more specialized than the other.

[Example:

```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 {}; // OK: Correspondence of [T, U, V] and [U, V, T]
// error: Vec not deduced
```

13.7.7 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. [Note: An alias template name is never deduced. —end note] [Example:

```cpp
template<class T> struct Alloc { /* ... */ }; // error: redefinition
template<class T> void process(Vec<T>& v) {
    // same as vector<int, Alloc<int>> v;
}
template<class T>
void process(vector<T, Alloc<T>>& w) {
    // error: redefinition
}
template<template<class> class TT>
void f(TT<int>); // 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:

    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:

    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:

    template <class T>
      using A = decltype([] { });     // A<int> and A<char> refer to different closure types

—end example]

13.7.8 Concept definitions [temp.concept]

A concept is a template that defines constraints on its template arguments.

collection:
  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:

    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; }

    template<C T>
      // C, as a type-constraint, constrains f2(T)
      T f2(T x) { return x; }

—end example]

A concept-definition shall appear at namespace scope (6.4.6).

A concept shall not have associated constraints (13.5.2).

A concept is not instantiated (13.9). [Note: A concept-id (13.3) is evaluated as an expression. A concept cannot be explicitly instantiated (13.9.2), explicitly specialized (13.9.3), or partially specialized. — end note]

The constraint-expression of a concept-definition is an unevaluated operand (7.2.3).

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

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 (13.8.2).
3. Names from scopes which are visible within the template definition.

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:

```cpp
// 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]

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. The usual qualified name lookup (6.5.3) is used to find the qualified-id even in the presence of typename. [Example:

```cpp
struct A {
    struct X { }; // error: no visible declaration of X
    int X;
};
struct B {
    struct X { }; // error: no visible declaration of 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]

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: The typename keyword is not permitted by the syntax of these constructs. — end note]
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, 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:

\[
\begin{align*}
\text{template<class } T \text{> } & \text{T::R f();} & \text{\textit{OK, return type of a function declaration at global scope}} \\
\text{template<class } T \text{> } & \text{void f(T::R);} & \text{\textit{ill-formed, no diagnostic required: attempt to declare a void variable template}} \\
\text{template<} & \text{class } T \text{> struct } S \{} \\
\text{using } & \text{Ptr = PtrTraits<T>::Ptr;} & \text{\textit{OK, in a defining-type-id}} \\
& \text{T::R f(T::P p) \{} & \text{\textit{OK, class scope}} \\
& \text{\quad return static_cast<T::R>(p);} & \text{\textit{OK, type-id of a static_cast}} \\
& \text{\}} & \text{\textit{OK, trailing-return-type}} \\
& \text{auto g()} & \text{\textit{variable pf of type void\* initialized with T::X}} \\
& \text{\rightarrow S<T>::Ptr;} & \text{\textit{error: T::X at block scope does not denote a type}} \\
& \text{\}} & \text{\textit{(attempt to declare a void variable)}} \\
\end{align*}
\]

—end example]

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. [Example:

\[
\begin{align*}
\text{template <class } T \text{> void f(int i) \{} & \text{\textit{expression, not the declaration of a variable i}} \\
& T::x * i; & \text{\textit{OK}} \\
\text{\}} & \text{\textit{error: Foo::x is a type}} \\
\end{align*}
\]

—end example]

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 (13.8.2.1). [Example:

\[
\begin{align*}
& \text{template<} & \text{typename } T \text{> } \text{void f();} & \text{\textit{OK, return type of a function template}} \\
& \text{template<} & \text{typename } T \text{> struct } S \{} & \text{\textit{OK, in a defining-type-id}} \\
& & \text{T::R f(T::P p) \{} & \text{\textit{OK, class scope}} \\
& & \text{\quad return static_cast<T::R>(p);} & \text{\textit{OK, type-id of a static_cast}} \\
& & \text{\}} & \text{\textit{OK, trailing-return-type}} \\
& \text{auto g()} & \text{\textit{variable pf of type void\* initialized with T::X}} \\
& \text{\rightarrow S<T>::Ptr;} & \text{\textit{error: T::X at block scope does not denote a type}} \\
& \text{\}} & \text{\textit{(attempt to declare a void variable)}} \\
\end{align*}
\]

\[136\) This includes friend function declarations.


```cpp
template<class T> struct A {
    typedef int B;
    B b; // OK, no typename required
};

— end example]

8 The validity of a template may be checked prior to any instantiation. [Note: 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 (8.5.1) 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: 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,

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: 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:

```cpp
int j;
template<class 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

When looking for the declaration of a name used in a template definition, the usual lookup rules (6.5.1, 6.5.2) are used for non-dependent names. The lookup of names dependent on the template parameters is postponed until the actual template argument is known (13.8.2). [Example:

```cpp
#include <iostream>
using namespace std;

template<class T> class Set {
    T* p;
    int cnt;
public:
    Set();
    Set<T>(const Set<T>&);
    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 must be 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 (13.8.2). —end example]

If a name does not depend on a template-parameter (as defined in 13.8.2), 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:

```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
}
```

```cpp
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]

[Note: 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]

13.8.1 Locally declared names  [temp.local]

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:

```cpp
template<template<class> class T> class A { }
template<class T> class Y;
template< 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:

```cpp
template <class T> struct Base {
    Base* p;
};

template <class T> struct Derived: public Base<int>, Base<char> {
    typename Derived::Base b;
    // error: ambiguous
    typename Derived::Base<double> d; // OK
};
```
—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:

```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]

5 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:

```cpp
template<class T> class X {
    X* p; // meaning X<T>
    X<T>* p2;
    X<int>* p3;
    ::X* p4; // error: missing template argument list
    ::::X does not refer to the injected-class-name
};
```
—end example]

6 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:

```cpp
template<class T, int i> class Y {
    int T; // error: template-parameter redeclared
};
```
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:**

```cpp
template<class T> struct A {
    struct B { /* ... */ };  
    typedef void C;
    void f(T);
    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:**

```cpp
namespace N {
    class C { };
    template<class T> class B {
        void f(T);
    };
}

template<class C> void N::B::f(C) {
    C b;              // C is the template parameter, not N::C
}
```

—end example—

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 (13.8.2.1), 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:**

```cpp
struct A {
    struct B { /* ... */ };  
    int a;
    int Y;
};

template<class B, class a> struct X : A {
    B b;              // A's B
    a b;              // error: A's a isn't a type name
};
```

—end example—
13.8.2 Dependent names

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 in this subclause.

In an expression of the form:

postfix-expression ( expression-list_opt )

where the postfix-expression is an unqualified-id, the unqualified-id denotes a dependent name if

1. any of the expressions in the expression-list is a pack expansion (13.7.3),
2. any of the expressions or braced-init-lists in the expression-list is type-dependent (13.8.2.2), or
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: Such names are unbound and are looked up at the point of the template instantiation (13.8.4.1) in both the context of the template definition and the context of the point of instantiation (13.8.4.2). — end note]

[Example:

```cpp
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 (13.8.2.1) 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:

```cpp
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:

```cpp
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

13.8.2.1 Dependent types

A name refers to the current instantiation if it is

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,

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),

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

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
th template parameter is a template parameter pack, the n
th template argument is a pack expansion (13.7.3) whose pattern is the name of the template parameter pack.

The template argument list of a primary template is a template argument list in which the n
th template argument has the value of the n
th template parameter of the class template. If the n
th template parameter is a template parameter pack (13.7.3), the n
th template argument is a pack expansion (13.7.3) whose pattern is the name of the template parameter pack.

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

1. it has the same type as the template parameter (ignoring cv-qualification) and
2. its initializer consists of a single identifier that names the template parameter or, recursively, such a variable.

[Note: Using a parenthesized variable name breaks the equivalence. — end note] [Example:

```
template <class T> class A {
    A* p1; // A is the current instantiation
    A<T>* p2; // A<T> 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);
```
A dependent base class is a base class that is a dependent type and is not the current instantiation. [Note: A base class can be the current instantiation in the case of a nested class naming an enclosing class as a base.]

```
template<class T> struct A {
  typedef int M;
  struct B {
    typedef void M;
    struct C;
  };
};
```

Example:

```
template<class T> struct A<T>::B::C : A<T> {
  M m; // OK, A<T>::M
};
```

—end example — end note

A name is a member of the current instantiation if it is

(5.1) — 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. [Note: This can only occur when looking up a name in a scope enclosed by the definition of a class template. — end note]

(5.2) — A qualified-id in which the nested-name-specifier refers to the current instantiation and 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. [Note: If no such member is found, and the current instantiation has any dependent base classes, then the qualified-id is a member of an unknown specialization; see below. — end note]

(5.3) — An id-expression denoting the member in a class member access expression (7.6.1.4) for which the type of the object expression is the current instantiation, and the id-expression, when looked up (6.5.5), refers to at least one member of a class that is the current instantiation or a non-dependent base class thereof. [Note: If no such member is found, and the current instantiation has any dependent base classes, then the id-expression is a member of an unknown specialization; see below. — end note]

Example:

```
template<class T> class A {
  static const int i = 5;
  int n1[i]; // i refers to a member of the current instantiation
  int n2[A::i]; // A::i refers to a member of the current instantiation
  int n3[A<T>::i]; // A<T>::i refers to a member of the current instantiation
  int f();
};
```

```
template<class T> int A<T>::f() {
  return i; // i refers to a member of the current instantiation
}
```

—end example

A name is a dependent member of the current instantiation if it is a member of the current instantiation that, when looked up, refers to at least one member of a class that is the current instantiation.

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.
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.

An id-expression denoting the member in a class member access expression (7.6.1.4) 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:

```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:

```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
    // does not occur in the template definition context; see 11.4.2
```

A type is dependent if it is

- a template parameter,
- a member of an unknown specialization,
- a nested class or enumeration that is a dependent member of the current instantiation,
- a cv-qualified type where the cv-unqualified type is dependent,
- a compound type constructed from any dependent type,
- an array type whose element type is dependent or whose bound (if any) is value-dependent.
— a function type whose exception specification is value-dependent,
— denoted by a 

\texttt{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 \[\text{[Note: This includes an injected-class-name (11.1) of a class template used without a template-argument-list. — end note]}\], or
— denoted by \texttt{decltype(expression)}, where \texttt{expression} is type-dependent (13.8.2.2).

\[\text{[Note: Because typedefs do not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent. — end note]}\]

\[13.8.2.2\] Type-dependent expressions

\[\text{[tempdepexpr]}\]

1 Except as described below, an expression is type-dependent if any subexpression is type-dependent.

2 \texttt{this} is type-dependent if the class type of the enclosing member function is dependent (13.8.2.1).

3 An \texttt{id-expression} is type-dependent if it is not a concept-id and it contains

\texttt{id-expression} \begin{enumerate}
\item an \texttt{identifier} associated by name lookup with one or more declarations declared with a dependent type,
\item an \texttt{identifier} associated by name lookup with a non-type \texttt{template-parameter} declared with a type that contains a placeholder type (9.2.8.5),
\item an \texttt{identifier} associated by name lookup with a variable declared with a type that contains a placeholder type (9.2.8.5) where the initializer is type-dependent,
\item an \texttt{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,
\item an \texttt{identifier} associated by name lookup with a structured binding declaration (9.6) whose \texttt{brace-or-equal-initializer} is type-dependent,
\item the \texttt{identifier \_\_func\_} (9.5.1), where any enclosing function is a template, a member of a class template, or a generic lambda,
\item a \texttt{template-id} that is dependent,
\item a \texttt{conversion-function-id} that specifies a dependent type, or
\item a \texttt{nested-name-specifier} or a \texttt{qualified-id} that names a member of an unknown specialization;
\end{enumerate}

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 (13.7.1.4). Expressions of the following forms are type-dependent only if the type specified by the \texttt{type-id}, \texttt{simple-type-specifier} or \texttt{new-type-id} is dependent, even if any subexpression is type-dependent:

\begin{verbatim}
  simple-type-specifier ( expression-list_opt )
::opt new new-placement_opt new-type-id new-initializeropt
dynamic_cast < type-id > ( expression )
static_cast < type-id > ( expression )
reinterpret_cast < type-id > ( expression )
(type-id) cast-expression
\end{verbatim}

4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

\begin{verbatim}
literal
sizeof unary-expression
sizeof ( type-id )
sizeof ... ( identifier )
alignof ( type-id )
typeid ( expression )
typeid ( type-id )
::opt delete cast-expression
typeid ( type-id )
::opt delete [ ] cast-expression
throw assignment-expression_opt
noexcept ( expression )
\end{verbatim}
A class member access expression (7.6.1.4) 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: In an expression of the form \texttt{x.y} or \texttt{xp->y} the type of the expression is usually the type of the member \texttt{y} of the class of \texttt{x} (or the class pointed to by \texttt{xp}). However, if \texttt{x} or \texttt{xp} refers to a dependent type that is not the current instantiation, the type of \texttt{y} is always dependent. If \texttt{x} or \texttt{xp} refers to a non-dependent type or refers to the current instantiation, the type of \texttt{y} is the type of the class member access expression. —end note]

A \textit{braced-init-list} is type-dependent if any element is type-dependent or is a pack expansion.

A \textit{fold-expression} is type-dependent.

### 13.8.2.3 Value-dependent expressions

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 \textit{id-expression} is value-dependent if:

1. it is a concept-id and any of its arguments are dependent,
2. it is type-dependent,
3. it is the name of a non-type template parameter,
4. it names a static data member that is a dependent member of the current instantiation and is not initialized in a \textit{member-declarator},
5. it names a static member function that is a dependent member of the current instantiation, or
6. 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 \textit{unary-expression} or \textit{expression} is type-dependent or the \textit{type-id} is dependent:

- \texttt{sizeof \ unary-expression}
- \texttt{sizeof ( \ type-id )}
- \texttt{typeid ( \ expression )}
- \texttt{typeid ( \ type-id )}
- \texttt{alignof ( \ type-id )}
- \texttt{noexcept ( \ expression )}

[Note: For the standard library macro \texttt{offsetof}, see 17.2. —end note]

Expressions of the following form are value-dependent if either the \textit{type-id} or \textit{simple-type-specifier} is dependent or the \textit{expression} or \textit{cast-expression} is value-dependent:

- \texttt{simple-type-specifier ( \ expression-list\opt )}
- \texttt{static\_cast < \ type-id > ( \ expression )}
- \texttt{const\_cast < \ type-id > ( \ expression )}
- \texttt{reinterpret\_cast < \ type-id > ( \ expression )}
- \texttt{( \ type-id ) cast-expression}

Expressions of the following form are value-dependent:

- \texttt{sizeof ... ( \ identifier )}
- \texttt{fold-expression}

An expression of the form \texttt{&qualified-id} where the \textit{qualified-id} names a dependent member of the current instantiation is value-dependent. An expression of the form \texttt{&cast-expression} is also value-dependent if evaluating \texttt{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.

### 13.8.2.4 Dependent template arguments

A type \textit{template-argument} is dependent if the type it specifies is dependent.

A non-type \textit{template-argument} is dependent if its type is dependent or the constant expression it specifies is value-dependent.
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.

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.

### 13.8.3 Non-dependent names

Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [Example:

```cpp
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]

### 13.8.4 Dependent name resolution

#### 13.8.4.1 Point of instantiation

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.

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.

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 *noexcept-specifier* immediately follows the namespace scope declaration or definition that requires the *noexcept-specifier*.

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, the point of instantiation of the specialization is the point of instantiation of the enclosing 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.

If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

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.

13.8.4.2 Candidate functions

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 (6.5.1, 6.5.2). [Note: For the part of the lookup using associated namespaces (6.5.2), function declarations found in the template instantiation context are found by this lookup, as described in 6.5.2. — 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:

Source file "X.h":

```cpp
namespace Q {
    struct X {}; // ADL in definition context finds Q::g_impl, g_impl not discarded
}
```

Source file "G.h":

```cpp
namespace Q {
    void g_impl(X, X);
}
```

Module interface unit of M1:

```cpp
module;
#include "X.h"
#include "G.h"
export module M1;
export template<typename T>
void g(T t) {
    g_impl(t, Q::X());
}
```

Module interface unit of M2:

```cpp
module;
#include "X.h"
export module M2;
import M1;
void h(Q::X x) {
    g(x); // OK
}
```

Example:

Module interface unit of Std:

```cpp
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
}
```

§ 13.8.4.2
Module interface unit of M:
export module M;
import Std;

struct S { /* ... */ }; void swap(S&, S&); // #1

void f(S* p, S* q)
{
    indirect_swap(p, q); // finds #1 via ADL in instantiation context
}

—end example]

[Example:]

Source file "X.h":
struct X { /* ... */ }; X operator+(X, X);

Module interface unit of F:
export module F;
export template<typename T>
void f(T t) {
    t + t;
}

Module interface unit of M:
module;
#include "X.h"
export module M;
import F;
void g(X x)
{
    f(x);
    // OK: instantiates f from F, operator+ is visible in instantiation context
}

—end example]

[Example:

Module interface unit of A:
export module A;
export template<typename T>
void f(T t) {
    cat(t, t);    // #1
dog(t, t);    // #2
}

Module interface unit of B:
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:
struct foo {
    friend int cat(foo, foo);
};
int dog(foo, foo);

Module interface unit of C1:
module;

§ 13.8.4.2
13.8.5 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 (11.9.3). Such names may be found under the rules for associated classes (6.5.2).\[137\] [Example:

```
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 \texttt{gcd} because \texttt{number<double>} is an associated class, making \texttt{gcd} visible in its namespace (global scope)
    b = gcd(3, 4);  // error: \texttt{gcd} is not visible
}
```
—end example]

---

137) Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.
13.9 Template instantiation and specialization

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`.

```c
// Example:

template<class T = int> struct A {
    static int x;
};
template<class U> void g(U) { }
template<> struct A<double> { } // specialize for T == double
template<> struct A<> { } // specialize for T == int
template<> void g(char) { } // specialize for U == char
template<> void g<int>(int) { } // specialize for U == int
template<> int A<char>::x = 0; // specialize for T == char

template<class T = int> struct B {
    static int x;
};
template<> int B<>::x = 1; // specialize for T == int
```

An instantiated template specialization can be either implicitly instantiated (13.9.1) for a given argument list or be explicitly instantiated (13.9.2). A specialization is a class, variable, function, or class member that is either instantiated (13.9.1) from a templated entity or is an explicit specialization (13.9.3) of a templated entity.

For a given template and a given set of template-arguments,

1. an explicit instantiation definition shall appear at most once in a program, ([5.1])
2. an explicit specialization shall be defined at most once in a program, as specified in 6.3, and ([5.2])
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. ([5.3])

An implementation is not required to diagnose a violation of this rule.

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: In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects that would normally not be accessible. — end note]**

Each class template specialization instantiated from a template has its own copy of any static members. **[Example:**

§ 13.9
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

If a function declaration acquired its function type through a dependent type (13.8.2.1) without using the syntactic form of a function declarator, the program is ill-formed. [Example:

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]

13.9.1 Implicit instantiation [temp.inst]

1 A template specialization $E$ is a declared specialization if there is a reachable explicit instantiation definition (13.9.2) or explicit specialization declaration (13.9.3) for $E$, or if there is a reachable explicit instantiation declaration for $E$ and $E$ is not

(1.1) — an inline function,
(1.2) — declared with a type deduced from its initializer or return value (9.2.8.5),
(1.3) — a potentially-constant variable (7.7), or
(1.4) — a specialization of a templated class.

[Note: 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: 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:

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 (13.8.4.1), the instantiation yields an incomplete class type (6.8). [Example:

template<class T> class X;
X<char> ch;     // error: incomplete type X<char>
— end example] [Note: 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]
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:

```cpp
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:

```cpp
template<class T, class U>
struct Outer {
    template<class X, class Y> struct Inner;
    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.

```cpp
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: 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]
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

7 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.

8 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:

```cpp
template<typename T> constexpr int f() { return T::value; }

template<bool B, typename T> void g(decltype(B ? f<T>() : 0));

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]

9 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:

```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]

10 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 (13.10.3).

11 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 (8.5.1), unless such instantiation is required. [Note: The instantiation of a generic lambda does not require instantiation of substatements of a constexpr if statement within its
compound-statement unless the call operator template is instantiated. — 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.

12 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:

```cpp
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]

13 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 (7.5.5.1) – 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`.

14 Each default argument is instantiated independently. [Example:

```cpp
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]

15 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.

16 [Note: 13.8.4.1 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:

```cpp
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]

The type-constraint 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 13.5.2 and 13.5.1.2 when determining whether the constraints are satisfied or as specified in 13.5.2 when comparing declarations. [Note: The satisfaction of constraints is determined during template argument deduction (13.10.2) and overload resolution (12.4). —end note] [Example:

```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>` (11.4.4.1), so there is no viable constructor for `s1`. —end example] [Example:

```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]

### 13.9.2 Explicit instantiation

1 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.

2 The syntax for explicit instantiation is:

```cpp
explicit-instantiation:
externopt 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.

3 An explicit instantiation shall not use a `storage-class-specifier` (9.2.1) 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.
4 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: The declaration may 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 (9.8.1), any namespace from its enclosing namespace set. [*Note: Regarding qualified names in declarators, see 9.3.3. —end note*] [*Example*:

```cpp
template<class T> class Array { void mf(); }
template class Array<char>;  // 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 (11.4.3), the program is ill-formed.

5 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: These declarations are required to have matching types as specified in 6.6, except as specified in 14.5. —end example*]

```cpp
template<typename T> T var = {};  // OK, instantiated variable has type float
template int var<float>();          // OK, absence of major array bound is permitted
template int *var<int>();           // error: instantiated variable has type int

```cpp
template<typename T> auto av = T();  // OK, variable with type int can be redeclared with type auto
```cpp
template<typename T> auto f() {}  // error: function with deduced return type
```cpp
template void f<int>();            // redeclared with non-deduced return type (9.2.8.5)

—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.

6 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.

7 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:

```c++
namespace N {
    template<class T> class Y { void mf() { }};
}
```
template class Y<int>; // error: class template Y not visible in the global namespace

using N::Y;
template class Y<int>; // error: explicit instantiation outside of the namespace of the template

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 (13.10.2). [Example:

```c++
template<class T> class Array { /* ... */};
template<class T> void sort(Array<T>& v) { /* ... */}
```

// instantiate sort(Array<int>&) – template-argument deduced
template void sort<>(Array<int>&);
—end example]

[Note: An explicit instantiation of a constrained template is required to satisfy that template’s associated
constraints (13.5.2). 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 (13.10.2.6) 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 (13.5.2, 13.5.1), except as described below.
[Note: 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 (11.4.6) 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 (13.9.1) 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: 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:

```c++
char* p = 0;
template<class T> T g(T x = &p) { return x; }
template int g<int>(int); // OK even though &p isn't an int.
—end example]
```
13.9.3 Explicit specialization

1. An explicit specialization of any of the following:
   
   (1.1) — function template
   (1.2) — class template
   (1.3) — variable template
   (1.4) — member function of a class template
   (1.5) — static data member of a class template
   (1.6) — member class of a class template
   (1.7) — member enumeration of a class template
   (1.8) — member class template of a class or class template
   (1.9) — member function template of a class or class template

   can be declared by a declaration introduced by `template<>`; that is:

   `template<> declaration`

   

   [Example:
    
    `template<class T> class stream;`
    
    `template<> class stream<char> { /* ... */ };`
    
    `template<class T> class Array { /* ... */ };`
    
    `template<class T> void sort(Array<T>& v) { /* ... */ }`
    
    `template<> void sort<char*>(Array<char*>&);`
    
    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. —end example]

2. An explicit specialization shall not use a `storage-class-specifier` (9.2.1) other than `thread_local`.

3. An explicit specialization may be declared in any scope in which the corresponding primary template may be defined (9.8.1.2, 11.4, 13.7.2).

4. A declaration of a function template, class template, or variable template being explicitly specialized shall precede the declaration of the explicit specialization. [Note: 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:
    
    `template<> class X<int> { /* ... */ }; // error: X not a template`
    
    `template<class T> class X;`
    
    `template<> class X<char*> { /* ... */ }; // OK: X is a template`
    
    — end example]

5. 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 (11.4.3), the program is ill-formed.

6. 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 definition is required. In this case, the definition of the class template explicit specialization shall be in scope at the point at which the member is defined. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization. Members of an explicitly specialized class template are defined
in the same manner as members of normal classes, and not using the `template<>` syntax. The same is true when defining a member of an explicitly specialized member class. However, `template<>` is used in defining a member of an explicitly specialized member class template that is specialized as a class template. [Example:

```c++
    template<class T> struct A {
        struct B {};  
        template<class U> struct C {};  
    };

    template<> struct A<int> {
        void f(int);
    };

    void h() {
        A<int> a;
        a.f(16);  // A<int>::f must be defined somewhere
    }

    // template<> not used for a member of an explicitly specialized class template
    void A<int>::f(int) { /* ... */ }

    template<> struct A<char>::B {
        void f();
    };
    // template<> also not used when defining a member of an explicitly specialized member class
    void A<char>::B::f() { /* ... */ }

    template<> template<class U> struct A<char>::C {
        void f();
    };
    // template<> is used when defining a member of an explicitly specialized member class template
    // specialized as a class template
    template<> template<class U> void A<char>::C<U>::f() { /* ... */ }

    template<> struct A<short>::B {
        void f();
    };
    template<> void A<short>::B::f() { /* ... */ }  // error: template<> not permitted

    template<> template<class U> struct A<short>::C {
        void f();
    };
    template<> template<class U> void A<short>::C<U>::f() { /* ... */ }  // error: template<> required

    —end example]

If 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:

```c++
    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>&), T is String
    }

    template<> void sort<String>(Array<String>& v);  // error: specialization after use of primary template
    template<> void sort<char>(Array<char>& v);  // 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);
template<class T> enum class A<T>::S : T { sT );
template<> enum A<char>::E : char { echar };
    // error: A<char>::E was instantiated when char was instantiated
    // error: A<char>::E was instantiated when char was instantiated

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.

A template explicit specialization is in the scope of the namespace in which the template was defined.
[Example:
namespace N {
    template<class T> class X { /* ... */};
    template<class T> class Y { /* ... */};

    template<> class X<int> { /* ... */}; // OK: specialization in same namespace
    template<> 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]

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:

    template<class T> class X;
    // X is a class template
    template<class T> class X<int>;

    X<int> * p; // OK: pointer to declared class X<int>
    X<int> x; // error: object of incomplete class X<int>
—end example]

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:

    template<class T> class Array { /* ... */};
    template<class T> void sort(Array<T>& v);

    // explicit specialization for sort(Array<int>&)
    // with deduced template-argument of type int
    template<> void sort(Array<int>&); // OK
—end example]

[Note: An explicit specialization of a constrained template is required to satisfy that template’s associated
constraints (13.5.2). 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

§ 13.9.3 411
of function templates, during template argument deduction (13.10.2.6) when one or more trailing template
arguments are left unspecified. —end note]

13 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 (13.7.6).

14 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:

```c
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }

template<> inline void f<int>(int) { /* ... */ } // OK: inline
template<> int g<int>(int) { /* ... */ } // OK: not inline
```
—end example]

15 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: The definition of a static data member of a template that requires default-initialization must use a
braced-init-list:

```c
template<> X Q<int>::x; // declaration
template<> X Q<int>::x(); // error: declares a function
template<> X Q<int>::x {}; // definition
```
—end note]

16 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:

```c
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(X1, X1);

// member template specialization
template<> template<T> void A<int>::g1(int, char); // X1 deduced as char

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }
```
—end example]

17 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:

```c
template<class T1> class A {
  template<class T2> class B {
    void mf();
```
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:

```cpp
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) { }
    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: Default function arguments may 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

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 13.10.2), or obtained from default template arguments.

Each function template specialization instantiated from a template has its own copy of any static variable. [Example:

```cpp
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 \( f<int>(\text{int}*) \) has a static variable \( s \) of type \( \text{int} \) and \( f<char*>(\text{char}**) \) has a static variable \( s \) of type \( \text{char} \). — end example

### 13.10.1 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.

\[\text{Example:}\]
```cpp
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

```cpp
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 (11.4.4, 6.5.3.1).

A template argument list may be specified when referring to a specialization of a function template

1. when a function is called,
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. in an explicit specialization,
4. in an explicit instantiation, or
5. in a friend declaration.

### 4
Trailing template arguments that can be deduced (13.10.2) or obtained from default template-arguments may be omitted from the list of explicit template-arguments. A trailing template parameter pack (13.7.3) 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:]

```cpp
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 (*)\(\text{bool}\)
    f<void>(f<int>);  // error: f<int> does not denote a single function template specialization
    int k = 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 (*)\(\text{bool}\), Z deduced as an empty sequence
}
```

— end example

Note: 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 (9.3.3.5) is visible that would otherwise be used. For example:

```cpp
template <class T> int f(T);  // #1
int f(int);  // #2
```
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:

```cpp
template<class X, class Y, class Z> X f(Y,Z);
```

```cpp
void g() {
    f<int, const char*, double>("aa", 3.0);
    f<int, const char*>("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
}
```

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: Template parameters do not participate in template argument deduction if they are explicitly specified. For example,

```cpp
template<class T> void f(T);
```

```cpp
class Complex {
    Complex(double);
};
```

```cpp
void g() {
    f<Complex>(1); // OK, means f<Complex>(Complex(1))
}
```

[Note: Because the explicit template argument list follows the function template name, and because constructor templates (11.4.4) are named without using a function name (6.5.3.1), 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:

```cpp
template<class ... Types> void f(Types ... values);
```

```cpp
void g() {
    f<int*, float*>(0, 0, 0); // Types deduced as the sequence int*, float*, int
}
```

—end example

### 13.10.2 Template argument deduction

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:

```cpp
void f(Array<dcomplex>& cv, Array<int>& ci) {
    sort(cv); // calls sort(Array<dcomplex>&)
    sort(ci); // calls sort(Array<int>&)
}
```

and

```cpp
void g(double d) {
    int i = convert<int>(d); // calls convert<int,double>(double)
    int c = convert<char>(d); // calls convert<char,double>(double)
}
```
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.

After this substitution is performed, the function parameter type adjustments described in 9.3.3.5 are performed. [Example: A parameter type of “void (const int, int[5])” becomes “void(*)(int,int*)”.] [Note: 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 example]

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:

```cpp
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);
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<const int>(1,0);
}
```
— end example]

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 (13.5.2), those constraints are checked for satisfaction (13.5.1). 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. [Note: The equivalent substitution in exception specifications is done only when the noexcept-specifier is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. —end note]

\[\text{Example:}\]

\begin{verbatim}
template <class T> struct A { using X = typename T::X; }; 
template <class T> typename T::X f(typename A<T>::X); 
template <class T> void f(...) { } 
template <class T> auto g(typename A<T>::X) -> typename T::X; 
template <class T> void g(...) { } 
template <class T> typename T::X h(typename A<T>::X); 
template <class T> auto h(typename A<T>::X) -> typename T::X; // redeclaration 
template <class T> void g(...) { } 

void x() {
  f<int>(0); // OK, substituting return type causes deduction to fail
  g<int>(0); // error, substituting parameter type instantiates A<int>
  h<int>(0); // ill-formed, no diagnostic required
}
\end{verbatim}

8 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. [Note: If no diagnostic is required, the program is still ill-formed. Access checking is done as part of the substitution process. —end note] 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. [Note: The substitution into types and expressions can result in effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such effects are not in the “immediate context” and can result in the program being ill-formed. —end note]

9 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: The intent is to avoid requiring implementations to deal with substitution failure involving arbitrary statements. [Example:

\begin{verbatim}
template <class T> auto f(T) -> decltype([]() { T::invalid; }()); 
void f(...); 
f(0); // error: invalid expression not part of the immediate context

template <class T, std::size_t = sizeof([]() { T::invalid; }())>
void g(T); 
g(...); 
g(0); // error: invalid expression not part of the immediate context

template <class T> 
auto h(T) -> decltype([x = T::invalid]() { });
void h(...);

h(0); // error: invalid expression not part of the immediate context

template <class T> 
auto i(T) -> decltype([]() -> typename T::invalid { });
void i(...);
i(0); // error: invalid expression not part of the immediate context

template <class T> 
auto j(T t) -> decltype([](auto x) -> decltype(x.invalid) { } (t)); // #1
\end{verbatim}\]
void j(...); // #2
j(0); // deduction fails on #1, calls #2
— end example] — end note]

Example:

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: Type deduction may 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:
    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:
    template <class T> int f(typename T::B*);
    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
(11.4.1) — the specified member is not a type where a type is required, or
(11.4.2) — the specified member is not a template where a template is required, or
(11.4.3) — the specified member is not a non-type where a non-type is required.
[Example:
    template <int I> struct X { };  
    template <template <class T> class> struct Z { };  
    template <class T> void f(typename T::Y*){}  
    template <class T> void g(X<T::N>*){}  
    template <class T> void h(Z<T::template TT>*){}  
    struct A {};
    struct B { int Y; };  
    struct C {  
        typedef int N;  
    };  
    struct D {  
        typedef int TT;  
    };  

    int main() {
        // Deduction fails in each of these cases:
        f<A>(0); // A does not contain a member Y
        f<B>(0); // The Y member of B is not a type
        g<C>(0); // The N member of C is not a non-type
        h<D>(0); // The TT member of D is not a template
    }
— end example]
— Attempting to create a pointer to reference type.

— Attempting to create a reference to `void`.

— Attempting to create “pointer to member of T” when T is not a class type. [Example:
  
  ```cpp
template <class T> int f(int T::*);
  int i = f<int>(0);
  
  — end example]
```

— Attempting to give an invalid type to a non-type template parameter. [Example:
  
  ```cpp
template <class T, T> struct S {};
template <class T> int f(S<T, T>()*);
struct X {};
int i0 = f<X>(0);
  
  — end example]
```

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [Example:
  
  ```cpp
template <class T, T*> int f(int);
int i2 = f<int,1>(0);  // can't conv 1 to int*
  
  — end example]
```

— Attempting to create a function type in which a parameter has a type of `void`, or in which the return type is a function type or array type.

— end note]

12 [Example: In the following example, assuming a `signed char` cannot represent the value 1000, a narrowing conversion (9.4.4) would be required to convert the template-argument of type `int` to `signed char`, therefore substitution fails for the second template (13.4.2).
  
  ```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
  
  template <class T> void j(T const(&)[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
  
  § 13.10.2.1 419
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
doeduced from second element, N deduced as 2
o({ &f1, static_cast<char(*)(char)>(&f4) });    // error: conflicting deductions for T

—end example— For a function parameter pack that occurs at the end of the parameter-declaration-list, deduction is performed for each remaining argument of the call, taking the type P of the declarator-id of the function parameter pack as the corresponding function template parameter type. Each deduction deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. When a function parameter pack appears in a non-deduced context (13.10.2.5), the type of that pack is never deduced. [Example:

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
gl(x, y, z);         // error: Types is not deduced
gl<int, int, int>(x, y, z);  // OK, no deduction occurs
}
—end example—

2 If P is not a reference type:

(2.1) — If A is an array type, the pointer type produced by the array-to-pointer standard conversion (7.3.2) is used in place of A for type deduction; otherwise,

(2.2) — If A is a function type, the pointer type produced by the function-to-pointer standard conversion (7.3.3) is used in place of A for type deduction; otherwise,

(2.3) — If A is a cv-qualified type, the top-level cv-qualifiers of A’s type are ignored for type deduction.

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:

```cpp
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&);
it n3 = g(i);        // calls g<volatile int>(const volatile int&)
```

—end example— A 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 (12.4.1.8)). 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:

```cpp
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.
    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{0, 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 example

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:

(4.1) — 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`.

(4.2) — 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 (7.3.13) and/or qualification conversion (7.3.5).

(4.3) — 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:

```
    template <typename... T> struct X;
    template <> struct X<> {};  // Note: 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
    template <typename... T> struct X;
    template <> struct X<> {};  // Note: 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
```

5 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: 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]

6 When `P` is a function type, function pointer type, or pointer-to-member-function type:

(6.1) — If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.

(6.2) — 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.

7 [Example:

```
    // Only one function of an overload set matches the call so the function parameter is a deduced context.
    template <class T> int f(T (*)(T));
    int g(int);
    int g(char);
    int i = f(g); // calls f(int (*)(int))
    -- end example
```

§ 13.10.2.1 421
8 [Example:]

// Ambiguous deduction causes the second function parameter to be a non-deduced context.

```c++
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))
```
—end example

9 [Example:]

// The overload set contains a template, causing the second function parameter to be a non-deduced context.

```c++
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))
```
—end example

10 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: 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:

```c++
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]

13.10.2.2 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 13.10.2.5. Otherwise, deduction is performed with empty sets of types P and A.

2 A placeholder type (9.2.8.5) 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.

13.10.2.3 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, 12.4.1.5, and 12.4.1.6 for the determination of that type) as described in 13.10.2.5.

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 (7.3.2) 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 (7.3.3) 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.
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.

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:

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).
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”.
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”.
4. The deduced \( A \) can be another pointer or pointer-to-member type that can be converted to \( A \) via a qualification conversion.

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced \( A \), the type deduction fails.

13.10.2.4 Deducing template arguments during partial ordering

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: The creation of the transformed type is described in 13.7.6.2. — 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:
1. In the context of a function call, the types used are those function parameter types for which the function call has arguments.\(^{138}\)
2. In the context of a call to a conversion function, the return types of the conversion function templates are used.
3. In other contexts (13.7.6.2) the function template’s function type is used.

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:
1. If \( P \) is a reference type, \( P \) is replaced by the type referred to.
2. 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:
1. If \( P \) is a cv-qualified type, \( P \) is replaced by the cv-unqualified version of \( P \).
2. 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 13.10.2.5. 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

\(^{138}\) Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.
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:

```c++
template<class... Args> void f(Args... args);  // #1
template<class T1, class... Args> void f(T1 a1, Args... args); // #2
```

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]

[9] 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):

(9.1) — 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,

(9.2) — 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.

10 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 \( F \).

11 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 \( G \).

12 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: A template parameter used in a non-deduced context is considered used. — end note] [Example:

```c++
template <class T> T f(int); // #1
```

void g() {
  f<int>(1); // calls #1
}

— end example]

13 [Note: 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:

```c++
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]

13.10.2.5 Deducing template arguments from a type [temp.deduct.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 \( \tilde{A} \)), 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:

1. A function type includes the types of each of the function parameters and the return type.
2. A pointer-to-member type includes the type of the class object pointed to and the type of the member pointed to.
3. A type that is a specialization of a class template (e.g., \( A\text{<int}> \)) includes the types, templates, and non-type values referenced by the template argument list of the specialization.
4. 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: Under 13.10.2.1, if \( P \) contains no \( \text{template-parameters} \) 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 \textit{nested-name-specifier} of a type that was specified using a \textit{qualified-id}.
2. The \textit{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:
   - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
   - no function matches the function parameter type, or
   - 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 (9.4.4) but the parameter does not have a type for which deduction from an initializer list is specified (13.10.2.1). [Example:
   ```cpp
template<class T> void g(T);
g({1,2,3}); // error: no argument deduced for T
   ``` — end example]
7. A function parameter pack that does not occur at the end of the \textit{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: If a type is specified as \( A\text{<T>::B<T2>} \), both \( T \) and \( T2 \) are non-deduced. Likewise, if a type is specified as \( A\text{<I+J>::X<T>} \), \( I, J, \) and \( T \) are non-deduced. If a type is specified as \( \text{void f(typename A<T>::B, A<T>}'} \), the \( T \) in \( A\text{<T>::B} \) is non-deduced but the \( T \) in \( A\text{<T>}' \) is deduced. — end example]

[Example: Here is an example in which different \( \text{parameter/argument} \) pairs produce inconsistent template argument deductions:
```cpp
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:

    template <class T, class U> void f( T (*)( T, U, U ) );
    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:

    template<class T> void f(const T*) { }
    int* p;
    void s() {
        f(p);    // calls 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:

    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>&)
    }

— end example]  

8 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  
T cv T  
T*  
T&  
T&&  
T[i]  
T[integer-constant]  
template-name<T>  (where template-name refers to a class template)

    type(T)  
    T()  
    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)
type[i]

where (T) represents a parameter-type-list (9.3.3.5) 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, <> 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 (13.10.2.4), 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:

```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 (9.3.3.5) 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 (13.10.2.2) or when deducing template arguments from a function declaration (13.10.2.6) 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 (13.10.2.1) 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: 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:

```c++
template <class T> void f(T&&);
template <> void f(int&) { }  // #1
```

—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 (13.10.2.4), 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:
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:
X<int> (*)(char[6])
is of the form
template-name<T> (*)(type[i])
which is a variant of
type (*)(T)
where type is X<int> and T is 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:
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 long from the
// template argument value in the type A<2>

—end example]

14 The type of N in the type T[N] is std::size_t. [Example:
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 std::size_t from the type int[42]

—end example]

15 [Example:
template<class T, T i> void f(int (&a)[i]);
int v[10];
void g() {
  f(v); // OK; T is std::size_t
}

—end example]

16 [Note: 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:
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); // OK: cannot deduce template-argument i
f2<10>(v);       // OK
f3(v);           // OK: i deduced as 10
}

—end note]

17 [Note: 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:

```cpp
template <int i> class A { /* ... */ };  // T is not deduced here
template <int i> void g(A<i+1>);        // but T is deduced here
void k() {
    A<1> a1;
    A<2> a2;
    g(a1);     // error: deduction fails for expression i+1
    g<0>(a1);  // OK
    f(a1, a2); // OK
}
```
—end example] —end note]

18 [Note: 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]

19 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.139 [Example:

```cpp
template<int i> class A { /* ... */ };  // T is not deduced here
template<short s> void f(A<s>);        // but T is deduced here
void k1() {
    A<1> a;
    f(a);    // error: deduction fails for conversion from int to short
    f<1>(a); // OK
}
```
—end example]

20 A template-argument can be deduced from a function, pointer to function, or pointer-to-member-function
type.

[Example:

```cpp
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
```

139) Although the template-argument corresponding to a template-parameter of type bool may be deduced from an array bound,
the resulting value will always be true because the array bound will be nonzero.

§ 13.10.2.5
void g(char, int);
void h(int, int, int);
void h(char, int);

int m() {
    f(kg);          // error: ambiguous
    f(kh);          // OK: void h(char, int) is a unique match
    f(kfoo);        // error: type deduction fails because foo is a template
}

—end example

21 A template type-parameter cannot be deduced from the type of a function default argument. [Example:

```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:

```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

23 [Note: Template argument deduction involving parameter packs (13.7.3) can deduce zero or more arguments for each parameter pack. —end note] [Example:

```cpp
template<class> struct X { };  
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...) > { };  
template<class T, class ... Types> struct Y { };  

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

13.10.2.6 Deducing template arguments from a function declaration  [temp.deduct.decl]

1 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 (13.9.2), explicit specializations (13.9.3), and certain friend declarations (13.7.4). This is also done to determine whether a deallocation function template specialization matches a placement operator new (6.7.5.42, 7.6.2.7). 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 7.6.2.7. The deduction is done as described in 13.10.2.5.

2 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 (13.7.6.2), deduction fails and, in the declaration cases, the program is ill-formed.
13.10.3 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 (13.10.2) 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 12.4.3.¹⁴⁰

Example:

```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: Here is an example involving conversions on a function argument involved in template-argument deduction:

```cpp
template<class T> struct B { /* ... */ };  
template<class T> struct D : public B<T> { /* ... */ };  
template<class T> void f(B<T>&);

void g(B<int>& bi, D<int>& di) {
    f(bi);        // f(bi)
    f(di);        // f(D<int>& di)
}
```

—end example

Example: Here is an example involving conversions on a function argument not involved in template-parameter deduction:

```cpp
template<class T> void f(T*,int);      // #1
template<class T> void f(T,char);      // #2

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:

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 12.4.3 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.
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*>, either implicitly or explicitly generated, is present in some translation unit. — 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: 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:
```
void f() {
    goto l1;  // error
    goto l2;  // error
    try {
        goto l1;  // OK
        goto l2;  // error
        l1: ;
    } catch (...) {
        l2: ;
        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:
```
lab: try {
    T1 t1;
    try {
        T2 t2;
        if (condition)
            goto lab;
    } catch(...) { /* handler 2 */ }
    } catch(...) { /* handler 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:

```cpp
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: An exception can be thrown from one of the following contexts: throw-expressions (7.6.18), allocation functions (6.7.5.4.1), dynamic_cast (7.6.1.6), typeid (7.6.1.7), new-expressions (7.6.2.7), and standard library functions (16.4.1.4). — end note] An object is passed and the type of that object determines which handlers can catch it. [Example:

```cpp
throw "Help!";
```

can be caught by a handler of const char* type:

```cpp
try {
  // ...
} catch(const char* p) {
  // handle character string exceptions here
}
```

and

```cpp
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:

```cpp
try {
  f(1.2);
} catch(Overflow& oo) {
  // handle exceptions of type Overflow here
}
—end example]

2 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.

3 Throwing an exception copy-initializes (9.4, 11.4.4.2) a temporary object, called the exception object. An value 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 (11.7.3), or a pointer to an incomplete type other than cv void the program is ill-formed.

§ 14.2
The memory for the exception object is allocated in an unspecified way, except as noted in 6.7.5.4.1. 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:

(4.1) 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;

(4.2) when an object of type std::exception_ptr (17.9.6) 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 (6.9.2.1). [Note: No other thread synchronization is implied in exception handling. — end note] The implementation may then deallocate the memory for the exception object; any such deallocation is done in an unspecified way. [Note: A thrown exception does not propagate to other threads unless caught, stored, and rethrown using appropriate library functions; see 17.9.6 and 32.9. — end note]

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 (11.10.5). The destructor is potentially invoked (11.4.6).

An exception is considered caught when a handler for that exception becomes active (14.4). [Note: An exception can have active handlers and still be considered uncaught if it is rethrown. — end note]

If the exception handling mechanism handling an uncaught exception (14.6.2) directly invokes a function that exits via an exception, the function std::terminate is called (14.6.1). [Example:

```c
struct C {
    C() { }
    C(const C&) {
        if (std::uncaught_exceptions()) {
            throw 0; // throw during copy to handler’s exception-declaration object (14.4)
        }
    }
};

int main() {
    try {
        throw C(); // calls std::terminate if construction of the handler’s exception-declaration object is not elided (11.10.5)
    } catch(C) { }
}
```

— end example] [Note: Consequently, destructors should generally catch exceptions and not let them propagate. — end note]

### 14.3 Constructors and destructors [except.ctor]

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 (8.7.3), 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:

```c
struct A {
};

struct Y { ~Y() noexcept(false) { throw 0; } };

A f() {
    try {
        A a;
        Y y;
        A b;
        return {};
    } catch(A e) { } // try block
}
```

§ 14.3 435
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]

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: 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.

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: If the object was allocated by a new-expression (7.6.2.7), the matching deallocation function (6.7.5.4.2), 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 void*, const void*, volatile void*, or const volatile 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

(3.1) — 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

(3.2) — the handler is of type cv T or cv T& and T is an unambiguous public base class of E, or

(3.3) — 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

(3.3.1) — a standard pointer conversion (7.3.11) not involving conversions to pointers to private or protected or ambiguous classes

(3.3.2) — a function pointer conversion (7.3.13)

(3.3.3) — a qualification conversion (7.3.5), or

(3.4) — the handler is of type ev T or const T& where T is a pointer or pointer-to-member type and E is std::nullptr_t.

[Note: 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:

class Matherr { /* ... */ virtual void vf(); }; 
class Overflow: public Matherr { /* ... */ }; 
class Underflow: public Matherr { /* ... */ }; 
class Zerodivide: public Matherr { /* ... */ }; 

void f() {
   try {
      g();
   }
}
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: 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: 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 (14.6.1).

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 (8.7.3) 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 8.7.3).

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;

   (15.2) --- 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.

16 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 [except.spec]

1 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`,

§ 14.5 437
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 (9.3.3.5).

    noexcept-specifier:
       noexcept ( constant-expression )
       noexcept

2 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).

3 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 (13.9.2) 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.

4 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:

```cpp
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]

5 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 (14.6.1). [Note: 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:

```cpp
extern void f();  // potentially-throwing

void g() noexcept {
    f();  // valid, even if f throws
    throw 42;  // valid, effectively a call to std::terminate
}
```

The call to f is well-formed even though, when called, f might throw an exception. —end example]

6 An expression E is potentially-throwing if

1 (6.1) E is a function call (7.6.1.2) whose postfix-expression has a function type, or a pointer-to-function type, with a potentially-throwing exception specification, or

2 (6.2) 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 (6.3) E is a throw-expression (7.6.18), or

4 (6.4) E is a dynamic_cast expression that casts to a reference type and requires a runtime check (7.6.1.6), or

5 (6.5) E is a typeid expression applied to a (possibly parenthesized) built-in unary * operator applied to a pointer to a polymorphic class type (7.6.1.7), or
any of the immediate subexpressions (6.9.1) of \( E \) is potentially-throwing.

7 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:

- a constructor selected by overload resolution in the implicit definition of the constructor for class \( X \) to initialize a potentially constructed subobject, or
- a subexpression of such an initialization, such as a default argument expression, or,
- for a default constructor, a default member initializer.

[Note: 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, 14.6.1). — end note]

8 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.

9 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.

10 A deallocation function (6.7.5.4.2) with no explicit `noexcept-specifier` has a non-throwing exception specification.

11 The exception specification for a comparison operator function (12.6.2) 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:

```c
struct A {
    A(int = (A(S), 0)) noexcept;
    A(const A&) noexcept;
    A(A&&) noexcept;
    ~A();
};
struct 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;
struct 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
    // D::D(D&&, int) potentially-throwing
};
```

Furthermore, if \( A::\sim A() \) 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]

12 An exception specification is considered to be `needed` when:

- 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);
- 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;

§ 14.5
— the exception specification is compared to that of another declaration (e.g., an explicit specialization or an overriding virtual function);
— the function is defined; or
— the exception specification is needed for a defaulted function that calls the function. [Note: 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 [except.special]

The function `std::terminate` (14.6.1) is used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself. The function `std::current_exception()` (17.9.6) and the class `std::nested_exception` (17.9.7) can be used by a program to capture the currently handled exception.

### 14.6.1 The `std::terminate` function [except.terminate]

In some situations exception handling must be abandoned for less subtle error handling techniques. [Note: These situations are:

— 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
— when the exception handling mechanism cannot find a handler for a thrown exception (14.4), or
— when the search for a handler (14.4) encounters the outermost block of a function with a non-throwing exception specification (14.5), or
— when the destruction of an object during stack unwinding (14.3) terminates by throwing an exception, or
— when initialization of a non-local variable with static or thread storage duration (6.9.3.3) exits via an exception, or
— when destruction of an object with static or thread storage duration exits via an exception (6.9.3.4), or
— when execution of a function registered with `std::atexit` or `std::at_quick_exit` exits via an exception (17.5), or
— when a throw-expression (7.6.18) with no operand attempts to rethrow an exception and no exception is being handled (14.2), or
— when the function `std::nested_exception::rethrow_nested` is called for an object that has captured no exception (17.9.7), or
— when execution of the initial function of a thread exits via an exception (32.4.2.2), or
— 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
— when the destructor or the move assignment operator is invoked on an object of type `std::thread` that refers to a joinable thread (32.4.2.3, 32.4.2.4), or
— when a call to a `wait()`, `wait_until()`, or `wait_for()` function on a condition variable (32.6.3, 32.6.4) fails to meet a postcondition.

— end note]
permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to the function `std::terminate`.

### 14.6.2 The `std::uncaught_exceptions()` function

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: 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, 17.9.6), it is considered uncaught from the point of rethrow until the rethrown exception is caught. The function `std::uncaught_exceptions()` (17.9.5) returns the number of uncaught exceptions in the current thread.
15  Preprocessing directives [cpp]

15.1  Preamble [cpp.pre]

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 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 white-space

§ 15.1
A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints:

1. 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 white space containing no new-line characters) or follows white space containing at least one new-line character, and is
   
   - a `#` preprocessing token, or
   - an `import` preprocessing token immediately followed on the same logical line by a header-name, `<`, identifier, string-literal, or `:` preprocessing token, or
   - a `module` preprocessing token immediately followed on the same logical line by an identifier, `;`, or `;` preprocessing token, or
   - 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.\[141\] [Note: 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]

```
# // preprocessing directive
module ; // preprocessing directive
export module leftpad; // 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; // preprocessing directive (ill-formed at phase 7)
import :: // not a preprocessing directive
import -> // not a preprocessing directive

— end example]
```

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.

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.

---

141) Thus, preprocessing directives are commonly called “lines”. These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in 15.6.2, 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 white-space 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 white-space 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: 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 [cpp.cond]

- 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 )

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.

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.

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.

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.

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 expansion.

---

142) 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.

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: 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>

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.

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).

Preprocessing directives of the forms

```c
#if constant-expression new-line group
#elif constant-expression new-line group
```

check whether the controlling constant expression evaluates to nonzero.

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.

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: 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]

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: 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: Thus, the constant expression in the following #if directive and if statement (8.5.1) is not guaranteed to evaluate to the same value in these two contexts:

```c
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

—end note] 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

§ 15.2
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.

**Example:** This demonstrates a way to include a library optional facility only if it is available:

```c
#include <optional>
#include <optional>
#include <optional>
#define have_optional 1
#endif
#else
#include <optional>
#include <optional>
#define have_optional 1
#endif
#if __has_include(<optional>)
#include <optional>
#elif __has_include(<optional>)
#include <optional>
#endif
#define have_optional 0
#endif
```

**Example:** This demonstrates a way to use the attribute `[[acme::deprecated]]` only if it is available.

```c
#define ATTR_DEPRECATED(msg) [[deprecated(msg)]]
```

---

### 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

```c
#include <h-char-sequence>
```

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

```c
#include "q-char-sequence"
```

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

```c
#include <h-char-sequence>
```

with the identical contained sequence (including `>` characters, if any) from the original directive.

---

14] As indicated by the syntax, a preprocessing token shall not follow a `#else` or `#endif` directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.
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. 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.

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.

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.

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

[Note: Although an implementation may provide a mechanism for making arbitrary source files available to the < > search, in general programmers should use the < > form for headers provided with the implementation, and the " form for sources outside the control of the implementation. For instance:

#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"

—end note]

[Example: This illustrates macro-replaced #include directives:

#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"

—end example]
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.

1 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.

2 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.

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. [Note: This makes the line no longer a directive so it is not removed at the end of phase 4. —end note] Additionally, in the second form of `pp-import`, a `header-name` token is formed as if the `header-name-tokens` were the `pp-tokens` of a `#include` directive. The `header-name-tokens` are replaced by the `header-name` token. [Note: This ensures that imports are treated consistently by the preprocessor and later phases of translation. —end note]

5 Each `#define` directive encountered when preprocessing each translation unit in a program results in a distinct macro definition. [Note: A predefined macro name (15.11) is not introduced by a `#define` directive. Implementations providing mechanisms to redefine additional macros are encouraged to not treat them as being introduced by a `#define` directive. —end note] Importing macros from a header unit makes macro definitions from a translation unit visible in other translation units. Each macro definition has at most one point of definition in each translation unit and at most one point of undefinition, as follows:

5.1 The point of definition of a macro definition within a translation unit is the point at which its `#define` directive occurs (in the translation unit containing the `#define` directive), or, if the macro name is not lexically identical to a keyword (5.11) or to the identifiers `module` or `import`, the first point of macro `import` of a translation unit containing a point of definition for the macro definition, if any (in any other translation unit).

5.2 The point of undefinition of a macro definition within a translation unit is the first point at which a `#undef` directive naming the macro occurs after its point of definition, or the first point of macro `import` of a translation unit containing a point of undefinition for the macro definition, whichever (if any) occurs first.

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). [Note: The relative order of `pp-imports` has no bearing on whether a particular macro definition is active. —end note]

8 [Example:

Importable header "a.h":

```
#define X 123 // #1
#define Y 45 // #2
#define Z a // #3
#undef X // point of undefinition of #1 in "a.h"
```

Importable header "b.h":

```
import "a.h"; // point of definition of #1, #2, and #3, point of undefinition of #1 in "b.h"
#define X 456 // OK, #1 is not active
#define Y 6 // error: #2 is active
```

Importable header "c.h":

```
#define Y 45 // #4
#define Z c // #5
```

Importable header "d.h":

```
import "a.h"; // point of definition of #1, #2, and #3, point of undefinition of #1 in "d.h"
import "c.h"; // point of definition of #4 and #5 in "d.h"
int a = Y; // OK, active macro definitions #2 and #4 are valid redefinitions
```
15.6 Macro replacement

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as an object-like macro (see below) may be redefined by another \texttt{\#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 \texttt{\#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.

There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

If the \texttt{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 \ldots). There shall exist a \texttt{)} preprocessing token that terminates the invocation.

The identifiers \texttt{\_\_VA\_ARGS\_} and \texttt{\_\_VA\_OPT\_} shall occur only in the \texttt{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 \texttt{\#define} is called the \texttt{macro name}. There is one name space for macro names. Any white-space 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 \texttt{\#} 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

\begin{verbatim}
\# define identifier replacement-list new-line
\end{verbatim}

defines an \texttt{object-like macro} that causes each subsequent instance of the macro name\textsuperscript{145} to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.\textsuperscript{146} The replacement list is then rescanned for more macro names as specified below.

\begin{verbatim}
[Example: The simplest use of this facility is to define a “manifest constant”, as in
#define TABSIZE 100
int table[TABSIZE];
\end{verbatim}

\textsuperscript{145}Since, by macro-replacement time, all character-literals and 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{146}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.
A preprocessing directive of the form

```
#define identifier lparen identifier-list opt replacement-list new-line
#define identifier lparen replacement-list new-line
#define identifier lparen replacement-list, ... ) replacement-list new-line
```

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 `#define` preprocessing directive. Each subsequent instance of the function-like macro name followed by a ( 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 ) 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 white-space 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, the behavior is undefined.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```
The parentheses ensure that the arguments and the resulting expression are bound properly.

If there is a ... immediately preceding the ) 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 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 ...).

### 15.6.1 Argument substitution

```
va-opt-replacement:
___VA_OPT___ ( pp-tokens_opt )
```

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 # or ## preprocessing token nor followed by a ## preprocessing token, the preprocessing tokens naming the parameter are replaced by a token sequence determined as follows:

1. **If the parameter is of the form `va-opt-replacement`, the replacement preprocessing tokens are the preprocessing token sequence for the corresponding argument, as specified below.**
2. **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.**

```
#define LPAREN() (  
#define G(Q) 42  
#define F(R, X, ...) ___VA_OPT___(G R X )
int x = F(LPAREN(), 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.

---

1. A *conditionally-supported-directive* is a preprocessing directive regardless of whether the implementation supports it.
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))

display("Flag");
display("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);

results in
fprintf(stderr, "Flag");
fprintf(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));

—end example

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 (15.6.3, 15.6.4). 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: The placemarker tokens are removed before stringization (15.6.2), and can be removed by rescanning and further replacement (15.6.4). —end note] [Example:

#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)

SDEF(foo); // replaced by S foo;
SDEF(bar, 1, 2); // replaced by S bar = { 1, 2 };

#define H1(X, ...) X __VA_OPT__(##) __VA_ARGS__ // error: ## may not appear at
          // the beginning of a replacement list (15.6.3)

#define H2(X, Y, ...) __VA_OPT__(X ## Y,) __VA_ARGS__
#define H3(X, ...) # __VA_OPT__(X##X X##X)
#define H4(X, ...) __VA_OPT__(a X ## X) ## b
#define H5A(...) __VA_OPT__()/**/__VA_OPT__()
#define H5B(X) a ## X ## b
#define H5C(X) H5B(X)

H5C(H5A()) // replaced by ab

—end example]
15.6.2 The \# operator

1 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.

2 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 white space 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.

15.6.3 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.

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 placemarks 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.

4 [Example: The sequence

```c
#define str(s)  # s
#define xstr(s) str(s)
#define debug(s, t) printf("%d, %s", x#s, x#t)
#define INCFILE(n) vers#n
#define glue(a, b) a##b
#define xglue(a, b) glue(a, b)
#define HIGHLOW "hello"
#define LOW LOW", world"

defined(1, 2);
#include xstr(INCFILE(2).h)
```

results in

```c
debug(1, 2);
```

```c
#include <stdio.h>

int main()
```

```c
{ printf("x\d\t\x\t\%s\%s\n", x1, x2);
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

```c
```

1488) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
or, after concatenation of the character string literals,

```c
printf("x1= \%d, x2= \%s", x1, x2);
fputs("strncmp(\"abc\0d\", \"abc\", ') == 0: \0\n", s);
#include "vers2.h"  
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional. — end example]

5 [Example: In the following fragment:

```c
#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:

```c
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: To illustrate the rules for placemarker preprocessing tokens, the sequence

```c
#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

```c
int j[] = { 123, 45, 67, 89,
  10, 11, 12, };  
```

— end example]

15.6.4 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.

2 [Example: The sequence

```c
#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define z z[0]
#define h g(~
#define m(a) a(w)
#define u 0,1
#define t(a) a
#define p() int
#define q(x) x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x*(3,4)-w) | h S & 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
\[ f(2 \times (y+1)) + f(2 \times (f(2 \times (z[0])))) \% f(2 \times (0)) + t(1); \]
\[ f(2 \times (2+(3,4)-0,1)) \| f(2 \times (-5)) \& f(2 \times (0,1))^m(0,1); \]
\[ \text{int } i[] = \{ 1, 23, 4, 5, \}; \]
\[ \text{char } c[2][6] = \{ "hello", "" \}; \]

—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.

15.6.5 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

\[ \# \text{undef } \text{identifier } \text{new-line} \]

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

\[ \# \text{line } \text{digit-sequence } \text{new-line} \]

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

\[ \# \text{line } \text{digit-sequence } " \text{s-char-sequence}_{\text{opt}} " \text{new-line} \]

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

\[ \# \text{line } \text{pp-tokens}_{\text{new-line}} \]

(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 [cpp.error]

1 A preprocessing directive of the form

\[ \# \text{error } \text{pp-tokens}_{\text{opt}} \text{new-line} \]

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 [cpppragma]

1 A preprocessing directive of the form

\[ \# \text{pragma } \text{pp-tokens}_{\text{opt}} \text{new-line} \]
causes the implementation to behave in an implementation-defined manner. The behavior might 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

```
# new-line
```

has no effect.

15.11 Predefined macro names

The following macro names shall be defined by the implementation:

```
__cplusplus
The integer literal 202002L. [Note: It is intended that future versions of this International Standard will replace the value of this macro with a greater value. — end note]
```

```
__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.
```

```
__FILE__
The presumed name of the current source file (a character string literal).
```

```
__LINE__
The presumed line number (within the current source file) of the current source line (an integer literal).
```

```
__STDC_HOSTED__
The integer literal 1 if the implementation is a hosted implementation or the integer literal 0 if it is not.
```

```
__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: Larger alignments will be passed to `operator new(std::size_t, std::align_val_t)`, etc. (7.6.2.7). — end note]
```

```
__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.
```

The names listed in Table 19.

The macros defined in Table 19 shall be defined to the corresponding integer literal. [Note: Future versions of this International Standard might replace the values of these macros with greater values. — end note]

<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>
</tbody>
</table>

149) The presumed source file name can be changed by the `#line` directive.
150) The presumed line number can be changed by the `#line` directive.
Table 19: Feature-test macros (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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_constrain</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>200710L</td>
</tr>
<tr>
<td>__cpp_ref_qualifiers</td>
<td>200710L</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_bindings</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>
<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:
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.

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 (6.7.5.4.3).

\_\_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 pre-defined macro names in this subclause, or the identifier \texttt{defined}, is the subject of a \texttt{define} or a \texttt{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 ( \texttt{string-literal} )

is processed as follows: The \texttt{string-literal} is destringized by deleting the \texttt{L} prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence \texttt{"} by a double-quote, and replacing each escape sequence \texttt{\\} 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 \textit{pp-tokens} in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

2 \textit{Example:}

\texttt{#pragma listing on \"..\listing.dir\"}

can also be expressed as:

\_Pragma ( \texttt{"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:

\texttt{#define LISTING(x) \_Pragma(listing on #x)}
\texttt{#define PRAGMA(x) \_Pragma(#x)}

\texttt{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 definitions (16.3), method of description (16.4), and organization (16.5.1) of the library. 16.5, 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 (7.6.2.7, 7.6.2.8) 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 (6.7.5.4), 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 \texttt{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 \texttt{restrict} qualifier) are the same unless otherwise stated.

16.3 Definitions

[library.c]

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 \texttt{restrict} qualifier) are the same unless otherwise stated.

16.3.1 arbitrary-positional stream

Stream (described in Clause 29) that can seek to any integral position within the length of the stream

[Note 1 to entry: Every arbitrary-positional stream is also a repositionable stream. — end note]

16.3.2 character

(Clause 21, Clause 28, Clause 29, and Clause 30) object which, when treated sequentially, can represent text

[Note 1 to entry: The term does not mean only \texttt{char}, \texttt{char8_t}, \texttt{char16_t}, \texttt{char32_t}, and \texttt{wchar_t} objects, but any value that can be represented by a type that provides the definitions specified in these Clauses. — end note]

16.3.3 character container type

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]

16.3.4 comparison function

Operator function (12.6) for any of the equality (7.6.10), relational (7.6.9), or three-way comparison (7.6.8) operators

16.3.5 component

Group of library entities directly related as members, parameters, or return types

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

16.3.6 constant subexpression

Expression whose evaluation as subexpression of a \texttt{conditional-expression} \texttt{CE} (7.6.16) would not prevent \texttt{CE} from being a core constant expression (7.7)
16.3.7 [defns.deadlock] 
deadlock
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

16.3.8 [defns.default.behavior.impl] 
default behavior
(implementation) specific behavior provided by the implementation, within the scope of the required behavior

16.3.9 [defns.default.behavior.func] 
default behavior
(specification) description of replacement function and handler function semantics

16.3.10 [defns.direct-non-list-init] 
direct-non-list-initialization
direct-initialization (9.4) that is not list-initialization (9.4.4)

16.3.11 [defns.expression-equivalent] 
expression-equivalent
expressions that all have the same effects, either are all potentially-throwing (14.5) or are all not potentially-
throwing, and either are all constant subexpressions or are all not constant subexpressions

[Example: For a value x of type 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]

16.3.12 [defns.handler] 
handler function
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]

16.3.13 [defns.order.ptr] 
implementation-defined strict total order over pointers
implementation-defined strict total ordering over all pointer values such that the ordering is consistent with
the partial order imposed by the builtin operators <, >, <=, >=, and <=>

16.3.14 [defns.iostream.templates] 
iostream class templates
templates, defined in Clause 29, that take two template arguments

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

16.3.15 [defns.modifier] 
modifier function
class member function (11.4.1) other than a constructor, assignment operator, or destructor that alters the
state of an object of the class

16.3.16 [defns.move.assign] 
move assignment
assignment of an rvalue of some object type to a modifiable lvalue of the same type

16.3.17 [defns.move.constr] 
move construction
direct-initialization of an object of some type with an rvalue of the same type

16.3.18 [defns.ntcts] 
NTCTS
sequence of values that have character type that precede the terminating null character type value charT()}
16.3.19 observer function

Class member function (11.4.1) 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 (11.4.2.1). — end note]

16.3.20 program-defined specialization

Explicit template specialization or partial specialization that is not part of the C++ standard library and not defined by the implementation.

16.3.21 program-defined type

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]

16.3.22 projection

Transformation that an algorithm applies before inspecting the values of elements.

[Example:]

```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:

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

— end example

16.3.23 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]

16.3.24 replacement function

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]

16.3.25 repositional stream

Stream (described in Clause 29) that can seek to a position that was previously encountered.

16.3.26 required behavior

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]
16.3.27 [defns.reserved.function]  
reserved function  
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]

16.3.28 [defns.stable]  
stable algorithm  
algorithm that preserves, as appropriate to the particular algorithm, the order of elements

[Note 1 to entry: Requirements for stable algorithms are given in 16.5.5.8. — end note]

16.3.29 [defns.traits]  
traits class  
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

16.3.30 [defns.valid]  
valid but unspecified state  
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: If an object \( x \) of type `std::vector<int>` is in a valid but unspecified state, \( x.empty() \) can be called unconditionally, and \( x.front() \) can be called only if \( x.empty() \) returns `false`. — end example]

16.4 Method of description [description]  

This subclause describes the conventions used to specify the C++ standard library. 16.4.1 describes the structure of the normative Clause 17 through Clause 32 and Annex D. 16.4.2 describes other editorial conventions.

16.4.1 Structure of each clause [structure]  

16.4.1.1 Elements [structure.elements]  

(1.1) — Summary
(1.2) — Requirements
(1.3) — Detailed specifications
(1.4) — References to the C standard library

16.4.1.2 Summary [structure.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.

2 The contents of the summary and the detailed specifications include:

(2.1) — macros
(2.2) — values
(2.3) — types and alias templates
(2.4) — classes and class templates
(2.5) — functions and function templates
(2.6) — objects and variable templates
(2.7) — concepts

---

151) 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.
16.4.1.3 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. Template arguments
2. Derived classes
3. Containers, iterators, and algorithms that meet an interface convention or model a concept

The string and iostream 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 type parameters.

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 16.4.2.2.

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.\(^{152}\)

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. [Example: The required < operator of the totally_ordered concept (18.5.4) does not meet the semantic requirements of that concept when operating on NaNs. — end example] This does not affect whether a type models the concept.

A declaration may explicitly impose requirements through its associated constraints (13.5.2). When the associated constraints refer to a concept (13.7.8), the semantic constraints specified for that concept are additionally imposed on the use of the declaration.

16.4.1.4 Detailed specifications

The detailed specifications each contain the following elements:

1. Name and brief description
2. Synopsis (class definition or function declaration, as appropriate)
3. Restrictions on template arguments, if any
4. Description of class invariants
5. Description of function semantics

Descriptions of class member functions follow the order (as appropriate):\(^{153}\)

1. Constructor(s) and destructor
2. Copying, moving & assignment functions
3. Comparison functions
4. Modifier functions

\(^{152}\) Although in some cases the code given is unambiguously the optimum implementation.

\(^{153}\) To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no “Comparison functions” subclause.
Descriptions of function semantics contain the following elements (as appropriate):\(^{154}\)

- **Constraints:** the conditions for the function’s participation in overload resolution (12.4). [Note: Failure to meet such a condition results in the function’s silent non-viability. — end note] [Example: An implementation might express such a condition via a constraint-expression (13.5.2). — end example]

- **Mandates:** the conditions that, if not met, render the program ill-formed. [Example: 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 (13.5.2) and also define the function as deleted. — end example]

- **Preconditions:** the conditions that the function assumes to hold whenever it is called.

- **Effects:** the actions performed by the function.

- **Synchronization:** the synchronization operations (6.9.2) applicable to the function.

- **Postconditions:** the conditions (sometimes termed observable results) established by the function.

- **Returns:** a description of the value(s) returned by the function.

- **Throws:** any exceptions thrown by the function, and the conditions that would cause the exception.

- **Complexity:** the time and/or space complexity of the function.

- **Remarks:** additional semantic constraints on the function.

- **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 (8.7.3) 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.\(^ {155}\)

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).

### 16.4.1.5 C library

Paragaphs labeled “See also” contain cross-references to the relevant portions of other standards (Clause 2).

### 16.4.2 Other conventions

This subclause describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (16.4.2.2), and member functions (16.4.2.3).

---

\(^{154}\) 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.

\(^{155}\) This simplifies the presentation of complexity requirements in some cases.
16.4.2.1 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&>()));
```

16.4.2.2 Type descriptions

16.4.2.2.1 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.

16.4.2.2.2 Exposition-only types

Several types defined in Clause 17 through Clause 32 and Annex D are only defined for the purpose of exposition. The declaration of such a type is followed by a comment ending in `exposition only`. [Example:

```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. 

16.4.2.2.3 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, ... };
```

§ 16.4.2.2.3
Here, the names \( C_0, C_1, \) etc. represent enumerated elements for this particular enumerated type. All such elements have distinct values.

### 16.4.2.2.4 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 {
    V_0 = 1 << 0,
    V_1 = 1 << 1,
    V_2 = 1 << 2,
    V_3 = 1 << 3,
    ...
};
```

```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 \( C_0, C_1, \) etc. 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:

- To **set** a value \( Y \) in an object \( X \) is to evaluate the expression \( X |= Y \).
- To **clear** a value \( Y \) in an object \( X \) is to evaluate the expression \( X &= ~Y \).
- The value \( Y \) is **set** in the object \( X \) if the expression \( X & Y \) is nonzero.

### 16.4.2.2.5 Character sequences

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 sentenced (int, const char*),\(^\text{158}\) or by a change to a locale object, as described in 28.3 and Clause 29.

---

A character sequence is an array object (9.3.3.4) \(\text{T A}[N]\), where \(\text{T}\) is any of the types char, unsigned char, or signed char (6.8.1), 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 \(\text{S}\) that points to its first element.

### 16.4.2.5.1 Byte strings

1. 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.\(^\text{159}\)

2. The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

3. The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

4. A static NTBS is an NTBS with static storage duration.\(^\text{160}\)

### 16.4.2.5.2 Multibyte strings

1. 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.\(^\text{161}\)

2. A static NTMBS is an NTMBS with static storage duration.

### 16.4.2.6 Customization Point Object types

1. 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.

2. The type of a customization point object, ignoring cv-qualifiers, shall model semiregular (18.6).

3. All instances of a specific customization point object type shall be equal (18.2).

4. The type \(\text{T}\) of a customization point object shall model invocable<const \(\text{T}\&, \text{Args}...>\) (18.7.2) when the types in \(\text{Args}...\) meet the requirements specified in that customization point object’s definition. When the types of \(\text{Args}...\) do not meet the customization point object’s requirements, \(\text{T}\) shall not have a function call operator that participates in overload resolution.

5. Each customization point object type constrains its return type to model a particular concept.

6. [Note: 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 (6.5.2). 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 (13.7.6.2) or more constrained (13.5.4) to be used by a customization point object. — end note]

### 16.4.2.3 Functions within classes

1. 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 generated by default (11.4.4.2, 11.4.5, 11.4.6). It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

\(^{158}\) declared in \(<\text{locale}\) (28.5.1).

\(^{159}\) Many of the objects manipulated by function signatures declared in \(<\text{cstring}\) (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.

\(^{160}\) A string-literal, such as "abc", is a static NTBS.

\(^{161}\) 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.
16.4.2.4 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:

\[ \text{streambuf* sb; // exposition only} \]

An implementation may use any technique that provides equivalent observable behavior.

16.5 Library-wide requirements

This subclause 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.

Within this subclause, 16.5.1 describes the library’s contents and organization, 16.5.2 describes how well-formed C++ programs gain access to library entities, 16.5.3 describes constraints on types and functions used with the C++ standard library, 16.5.4 describes constraints on well-formed C++ programs, and 16.5.5 describes constraints on conforming implementations.

16.5.1 Library contents and organization

16.5.1.1 Library contents

The C++ standard library provides definitions for the entities and macros described in the synopses of the C++ standard library headers (16.5.1.2), 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.

16.5.1.2 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: Importable C++ library headers can be imported as module units (10.3). —end note] [Example:

```
import <vector>; // imports the <vector> header unit
std::vector<int> vi; // OK
```

—end example]

Except as noted in Clause 16 through Clause 32 and Annex D, the contents of each header \( \text{c} \) name is the same as that of the corresponding header \( \text{name.h} \) as specified in the C standard library (Clause 2). In the

162) The C standard library headers (D.9) also define names within the global namespace, while the C++ headers for C library facilities (16.5.1.2) may also define names within the global namespace.

163) This gives implementers freedom to use inline namespaces to support multiple configurations of the library.

164) 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).

165) It is intentional that there is no C++ header for any of these C headers: <stdatomic.h>, <stdnoreturn.h>, <threads.h>.
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: 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.\(^6\)

8 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.\(^7\)

9 D.9, C standard library headers, describes the effects of using the `name.h` (C header) form in a C++ program.\(^8\)

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`.)

11 Table 23 lists the Annex K names that may be declared in some header. These names are also subject to the restrictions of 16.5.4.3.2.

---

### 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>` | `<iomanip>` | `<random>` | `<thread>` |
| `<bitset>` | `<ios>` | `<ranges>` | `<tuple>` |
| `<charconv>` | `<iosfwd>` | `<ratio>` | `<typeindex>` |
| `<chrono>` | `<iostream>` | `<regex>` | `<typeinfo>` |
| `<codecvt>` | `<iostream>` | `<scoped_allocator>` | `<type_traits>` |
| `<compare>` | `<iterator>` | `<semaphore>` | `<unordered_map>` |
| `<complex>` | `<latch>` | `<set>` | `<unordered_set>` |
| `<concept>` | `<limits>` | `<shared_mutex>` | `<utility>` |
| `<condition_variable>` | `<list>` | `<source_location>` | `<valarray>` |
| `<coroutine>` | `<locale>` | `<span>` | `<variant>` |
| `<deque>` | `<local>` | `<sstream>` | `<vector>` |
| `<exception>` | `<map>` | `<sstream>` | `<version>` |
| `<execution>` | `<memory>` | `<stack>` |  |
| `<filesystem>` | `<memory_resource>` | `<stdexcept>` |  |
| `<format>` | `<mutex>` | `<stop_token>` |  |
| `<forward>` | `<new>` | `<streambuf>` |  |

### Table 22: C++ headers for C library facilities

| `<cassert>` | `<cfenv>` | `<climits>` | `<csetjmp>` | `<cstddef>` | `<cstdlib>` | `<cstdint>` | `<cstdio>` | `<cstring>` | `<ctime>` |
| `<cctype>` | `<cfloat>` | `<cinttypes>` | `<cwchar>` | `<cwctype>` | `<csetjmp>` | `<csignal>` | `<cstdarg>` | `<cstddef>` | `<cwctype>` |

---

166) 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.

167) In particular, including the standard header `<iso646.h>` has no effect.

168) 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.
### Freestanding implementations

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. A freestanding implementation has an implementation-defined set of headers. This set shall include at least the headers shown in Table 24.

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.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>17.3</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</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>17.5</td>
<td><code>&lt;cstdliblib&gt;</code></td>
</tr>
<tr>
<td>17.6</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>17.7</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>17.8</td>
<td><code>&lt;source_location&gt;</code></td>
</tr>
<tr>
<td>17.9</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>17.10</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>17.11</td>
<td><code>&lt;compare&gt;</code></td>
</tr>
<tr>
<td>17.12</td>
<td><code>&lt;coroutine&gt;</code></td>
</tr>
<tr>
<td>17.13</td>
<td><code>&lt;cstdarg&gt;</code></td>
</tr>
<tr>
<td>Clause 18</td>
<td><code>&lt;concepts&gt;</code></td>
</tr>
<tr>
<td>20.15</td>
<td><code>&lt;type_traits&gt;</code></td>
</tr>
<tr>
<td>26.5</td>
<td><code>&lt;bit&gt;</code></td>
</tr>
<tr>
<td>Clause 31</td>
<td><code>&lt;atomic&gt;</code></td>
</tr>
</tbody>
</table>
16.5.2 Using the library

16.5.2.1 Overview

Subclause 16.5.2 describes how a C++ program gains access to the facilities of the C++ standard library. 16.5.2.2 describes effects during translation phase 4, while 16.5.2.3 describes effects during phase 8 (5.2).

16.5.2.2 Headers

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).

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>` (19.3.1) or `<assert.h>` (D.9) depends each time on the lexically current definition of `NDEBUG`.

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.

16.5.2.3 Linkage

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).

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.

Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

See also replacement functions (16.5.4.6), runtime changes (16.5.4.7).

16.5.3 Requirements on types and expressions

16.5.3.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 16.5.3.2 describes the requirements on swappable types and swappable expressions. 16.5.3.3 describes the requirements on pointer-like types that support null values. 16.5.3.4 describes the requirements on hash function objects. 16.5.3.5 describes the requirements on storage allocators.

16.5.3.1 Template argument requirements

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, T is an object or reference type to be supplied by a C++ program instantiating a template; a, b, and c are values of type (possibly `const`) T; s and t are modifiable lvalues of type T; u denotes an identifier; rv is an rvalue of type T; and v is an lvalue of type (possibly `const`) T or an rvalue of type `const` T.

In general, a default constructor is not required. Certain container class member function signatures specify T()` as a default argument. T()` shall be a well-defined expression (9.4) if one of those signatures is called using the default argument (9.3.3.6).

16.5.3.2 Swappable requirements

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.

An object t is swappable with an object u if and only if:

— the expressions `swap(t, u)` and `swap(u, t)` are valid when evaluated in the context described below, and

(2.1) This is the same as the C standard library.

(170) 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>
</table>
| `a == b`   | convertible to `bool` | `==` is an equivalence relation, that is, it has the following properties:  
- For all `a`, `a == a`.  
- If `a == b`, then `b == a`.  
- If `a == b` and `b == c`, then `a == c`. |

### Table 26: Cpp17LessThanComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &lt; b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>&lt;</code> 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><code>T t;</code></td>
<td>object <code>t</code> is default-initialized</td>
</tr>
<tr>
<td><code>T u{};</code></td>
<td>object <code>u</code> is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td><code>T()</code></td>
<td>an object of type <code>T</code> is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td><code>T{}</code></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><code>T u = rv;</code></td>
<td><code>u</code> is equivalent to the value of <code>rv</code> before the construction</td>
</tr>
<tr>
<td><code>T(rv)</code></td>
<td><code>T(rv)</code> is equivalent to the value of <code>rv</code> before the construction</td>
</tr>
<tr>
<td><code>rv</code>'s state is unspecified. [Note: <code>rv</code> must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether <code>rv</code> has been moved from or not. —end note]</td>
<td></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><code>T u = v;</code></td>
<td>the value of <code>v</code> is unchanged and is equivalent to <code>u</code></td>
</tr>
<tr>
<td><code>T(v)</code></td>
<td>the value of <code>v</code> is unchanged and is equivalent to <code>T(v)</code></td>
</tr>
</tbody>
</table>

### Table 30: Cpp17MoveAssignable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Return value</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t = rv</code></td>
<td><code>T&amp;</code></td>
<td><code>t</code></td>
<td>If <code>t</code> and <code>rv</code> do not refer to the same object, <code>t</code> is equivalent to the value of <code>rv</code> before the assignment</td>
</tr>
<tr>
<td><code>rv</code>'s state is unspecified. [Note: <code>rv</code> must still meet the requirements of the library component that is using it, whether or not <code>t</code> and <code>rv</code> refer to the same object. The operations listed in those requirements must work as specified whether <code>rv</code> has been moved from or not. —end note]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 31: Cpp17CopyAssignable requirements (in addition to Cpp17MoveAssignable)

<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: Array types and non-object types are not Cpp17Destructible. —end note)

(2.2) these expressions have the following effects:

1. 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:

   - the two `swap` function templates defined in `<utility>` (20.2) and
   - the lookup set produced by argument-dependent lookup (6.5.2).

   (Note: 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)

(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.

3. 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.

4. An rvalue or lvalue t is swappable if and only if t is swappable with any rvalue or lvalue, respectively, of type T.

5. An rvalue or lvalue t is swappable if and only if t is swappable with any rvalue or lvalue, respectively, of type T.

6. [Example: User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

   ```
   #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.
   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
   }

   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
   ```

§ 16.5.3.2 473
```cpp
// conditions for fundamental types

void swap(Proxy p, &x) { swap(x, p); }  // satisfy symmetry constraint

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);
}

— end example

16.5.3.3 Cpp17NullablePointer requirements [nullablepointer.requirements]
1 A Cpp17NullablePointer type is a pointer-like type that supports null values. A type P meets the Cpp17NullablePointer requirements if:

(1.1) — P meets the Cpp17EqualityComparable, Cpp17DefaultConstructible, Cpp17CopyConstructible, Cpp17CopyAssignable, and Cpp17Destructible requirements,
(1.2) — lvalues of type P are swappable (16.5.3.2),
(1.3) — the expressions shown in Table 33 are valid and have the indicated semantics, and
(1.4) — P meets all the other requirements of this subclause.

2 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: Operations involving indeterminate values may cause undefined behavior. — end note]

3 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.

4 No operation which is part of the Cpp17NullablePointer requirements shall exit via an exception.

5 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.

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 = 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>

16.5.3.4 Cpp17Hash requirements [hash.requirements]
1 A type H meets the Cpp17Hash requirements if:

(1.1) — it is a function object type (20.14),
(1.2) — it meets the Cpp17CopyConstructible (Table 29) and Cpp17Destructible (Table 32) requirements, and
(1.3) — the expressions shown in Table 34 are valid and have the indicated semantics.

2 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.

§ 16.5.3.4 474
Table 34: Cpp17Hash requirements  [tab:cpp17.hash]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(k)</td>
<td>size_t</td>
<td>The value returned shall depend only on the argument k for the duration of the program. [Note: 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] [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&lt;size_t&gt;::max(). — end note]</td>
</tr>
<tr>
<td>h(u)</td>
<td>size_t</td>
<td>Shall not modify u.</td>
</tr>
</tbody>
</table>

16.5.3.5 Cpp17Allocator requirements  [allocator.requirements]

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.

Table 35: Descriptive variable definitions  [tab:allocator.req.var]

<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&amp;&amp;</td>
</tr>
</tbody>
</table>

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  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::pointer</code></td>
<td><code>T*</code></td>
<td><code>X::pointer</code> is convertible to <code>X::const_pointer</code></td>
<td></td>
</tr>
<tr>
<td><code>X::const_pointer</code></td>
<td><code>X::pointer</code></td>
<td><code>X::const_pointer</code></td>
<td></td>
</tr>
<tr>
<td><code>X::void_pointer</code></td>
<td><code>X::pointer</code> is convertible to <code>X::void_pointer</code>, <code>X::void_pointer</code> and <code>Y::void_pointer</code> are the same type.</td>
<td><code>pointer_traits&lt;X::const_pointer&gt;::rebind&lt;const T&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>Y::void_pointer</code></td>
<td><code>X::const_pointer</code></td>
<td><code>pointer_traits&lt;X::pointer&gt;::rebind&lt;void&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>X::const_void_pointer</code></td>
<td><code>X::pointer</code>, <code>X::const_pointer</code>, and <code>X::void_pointer</code> are convertible to <code>X::const_void_pointer</code> and <code>Y::const_void_pointer</code> are the same type.</td>
<td><code>pointer_traits&lt;X::pointer&gt;::rebind&lt;const void&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>Y::const_void_pointer</code></td>
<td><code>X::const_pointer</code></td>
<td><code>pointer_traits&lt;X::pointer&gt;::rebind&lt;const void&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>X::value_type</code></td>
<td>Identical to <code>T</code></td>
<td><code>T</code></td>
<td></td>
</tr>
<tr>
<td><code>X::size_type</code></td>
<td><code>unsigned integer type</code></td>
<td><code>a type that can represent the size of the largest object in the allocation model</code></td>
<td><code>make_unsigned_t&lt;X::difference_type&gt;</code></td>
</tr>
<tr>
<td><code>X::difference_type</code></td>
<td><code>signed integer type</code></td>
<td><code>a type that can represent the difference between any two pointers in the allocation model</code></td>
<td><code>pointer_traits&lt;X::pointer&gt;::difference_type</code></td>
</tr>
<tr>
<td><code>typename X::template rebind&lt;U&gt;::other</code></td>
<td><code>Y</code></td>
<td><code>For all </code>U<code>(including</code>T<code>), </code>Y::template rebind&lt;T&gt;::other<code>is</code>X<code>.</code></td>
<td><code>See Note A, below.</code></td>
</tr>
<tr>
<td><code>*p</code></td>
<td><code>T&amp;</code></td>
<td><code>*q</code> refers to the same object as <code>*p</code>.</td>
<td></td>
</tr>
<tr>
<td><code>*q</code></td>
<td><code>const T&amp;</code></td>
<td><code>p-&gt;m</code></td>
<td><code>type of </code>T::m<code> </code>Preconditions: <code>(*p).m</code> is well-defined.`</td>
</tr>
<tr>
<td><code>p-&gt;m</code></td>
<td><code>type of </code>T::m`</td>
<td><code>equivalent to</code></td>
<td><code>(*p).m</code></td>
</tr>
<tr>
<td><code>q-&gt;m</code></td>
<td><code>type of </code>T::m`</td>
<td><code>Preconditions: </code>(*q).m<code> is well-defined.</code></td>
<td><code>equivalent to</code></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></td>
<td><code>== </code>p`</td>
</tr>
<tr>
<td><code>static_cast&lt;X::const_pointer&gt;(x)</code></td>
<td><code>X::const_pointer</code></td>
<td><code>static_cast&lt;X::const_pointer&gt;(x)</code></td>
<td><code>== </code>q`</td>
</tr>
<tr>
<td><code>pointer_traits&lt;X::pointer_to&gt;(r)</code></td>
<td><code>X::pointer</code></td>
<td><code>same as </code>p`</td>
<td></td>
</tr>
</tbody>
</table>
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.allocate(n)</td>
<td>X::pointer</td>
<td>Memory is allocated for an array of n T and such an object is created but array elements are not constructed. [Example: When using storage denoted by some pointer value p, <code>launder(reinterpret_cast&lt;T*&gt;(new (p) byte[n * sizeof(T)]))</code> can be used to implicitly create a suitable array object and obtain a pointer to it. — end example] <code>allocate</code> may throw an appropriate exception. [Note: If n == 0, the return value is unspecified. — end note]</td>
<td>a.allocate(n)</td>
</tr>
<tr>
<td>a.allocate(n, y)</td>
<td>X::pointer</td>
<td>Same as a.allocate(n). The use of y is unspecified, but it is intended as an aid to locality.</td>
<td>a.allocate(n)</td>
</tr>
<tr>
<td>a.deallocate(p,n)</td>
<td>(not used)</td>
<td>Preconditions: p is a value returned by an earlier call to <code>allocate</code> that has not been invalidated by an intervening call to <code>deallocate</code>. n matches the value passed to <code>allocate</code> to obtain this memory. Throws: Nothing.</td>
<td></td>
</tr>
<tr>
<td>a.max_size()</td>
<td>X::size_type</td>
<td>the largest value that can meaningfully be passed to <code>X::allocate()</code></td>
<td>numeric_limits&lt;size_type&gt;::max() / sizeof(value_type)</td>
</tr>
<tr>
<td>a1 == a2</td>
<td>bool</td>
<td>Returns true only if storage allocated from each can be deallocated via the other. <code>operator==</code> shall be reflexive, symmetric, and transitive, and shall not exit via an exception.</td>
<td></td>
</tr>
<tr>
<td>a1 != a2</td>
<td>bool</td>
<td>same as !(a1 == a2)</td>
<td></td>
</tr>
<tr>
<td>a == b</td>
<td>bool</td>
<td>same as a == Y::rebind&lt;T&gt;::other(b)</td>
<td></td>
</tr>
<tr>
<td>a ! = b</td>
<td>bool</td>
<td>same as !(a == b)</td>
<td></td>
</tr>
<tr>
<td>X u(a); X u = a;</td>
<td></td>
<td>Shall not exit via an exception. <code>Postconditions: u == a</code></td>
<td></td>
</tr>
<tr>
<td>X u(b);</td>
<td></td>
<td>Shall not exit via an exception. <code>Postconditions: Y(u) == b, u == X(b)</code></td>
<td></td>
</tr>
<tr>
<td>X u(std::move(a)); X u = std::move(a);</td>
<td></td>
<td>Shall not exit via an exception. <code>Postconditions: The value of a is unchanged and is equal to u.</code></td>
<td></td>
</tr>
</tbody>
</table>

171) 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>X u(std::move(b));</td>
<td></td>
<td>Shall not exit via an exception. Postconditions: u is equal to the prior value of X(b).</td>
<td></td>
</tr>
<tr>
<td>a.construct(c, args)</td>
<td>(not used)</td>
<td>Effects: 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>Effects: Destroys the object at c</td>
<td>destroy_at(c)</td>
</tr>
<tr>
<td>a.select_on_container_copy_construction()</td>
<td>X</td>
<td>Typically returns either a or X().</td>
<td>return a;</td>
</tr>
<tr>
<td>X::propagate_on_container_copy_assignment</td>
<td></td>
<td>Identical to or derived from true_type or false_type</td>
<td><code>false_type</code></td>
</tr>
<tr>
<td>X::propagate_on_container_move_assignment</td>
<td></td>
<td>Identical to or derived from true_type or false_type</td>
<td><code>false_type</code></td>
</tr>
<tr>
<td>X::propagate_on_container_swap</td>
<td>Identical to or derived from true_type or false_type</td>
<td><code>false_type</code></td>
<td></td>
</tr>
<tr>
<td>X::is_always_equal</td>
<td>Identical to or derived from true_type or false_type</td>
<td><code>true_type</code> only if the expression a1 == a2 is guaranteed to be true for any two (possibly const) values a1, a2 of type X.</td>
<td><code>is_empty&lt;X&gt;::type</code></td>
</tr>
</tbody>
</table>

3 Note A: The member class template `rebind` in the table above is effectively a typedef template. [Note: In general, if the name `Allocator` is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<T>::value_type` 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 zero or more type arguments, and `Allocator` does not supply a `rebind` member template, the standard `allocator_traits` template uses `SomeAllocator<U, 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`, values of type X shall be swappable (16.5.3.2) and the `swap` operation shall not throw exceptions.

5 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 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` (23.3.5.6) and the additional requirement that, when a and (a + n) are dereferenceable pointer values for some integral value n,

\[
\text{addressof}(*a + n) = \text{addressof}(\ast a) + n
\]

is true.

6 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 $x_1$ and $x_2$ can be explicitly converted to the two corresponding objects $px_1$ and $px_2$ of type $X::\text{const\_pointer}$, using a sequence of static\_casts using only these four types, and the expression $px_1 == px_2$ evaluates to true.

Let $w_1$ and $w_2$ denote objects of type $X::\text{void\_pointer}$. Then for the expressions

- $w_1 == w_2$
- $w_1 != w_2$

either or both objects may be replaced by an equivalently-valued object of type $X::\text{const\_void\_pointer}$ with no change in semantics.

Let $p_1$ and $p_2$ denote objects of type $X::\text{pointer}$. Then for the expressions

- $p_1 == p_2$
- $p_1 != p_2$
- $p_1 < p_2$
- $p_1 <= p_2$
- $p_1 > p_2$
- $p_1 >= p_2$
- $p_1 - p_2$

either or both objects may be replaced by an equivalently-valued object of type $X::\text{const\_pointer}$ with no change in semantics.

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.

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: Additionally, the member function allocate for that type may fail by throwing an object of type bad\_alloc. — end note]

[Example: The following is an allocator class template supporting the minimal interface that meets the requirements of Table 36:

```cpp
template<class Tp>
struct SimpleAllocator {
  typedef Tp value_type;
  SimpleAllocator(ctor args);  // constructor
  template<class T> SimpleAllocator(const SimpleAllocator<T>& other);  // copy constructor

  [[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>&);

— end example]

16.5.3.5.1 Allocator completeness requirements [allocator.requirements.completeness]

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.

16.5.4 Constraints on programs [constraints]

16.5.4.1 Overview [constraints.overview]

Subclause 16.5.4 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 (16.5.4.2.1), its use of various reserved names (16.5.4.3), its use of headers (16.5.4.4), its use of standard library classes as base
classes (16.5.4.5), its definitions of replacement functions (16.5.4.6), and its installation of handler functions during execution (16.5.4.7).

16.5.4.2 Namespace use

16.5.4.2.1 Namespace std

1 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.

2 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.

3 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.

4 The behavior of a C++ program is undefined if it declares

(a) an explicit specialization of any member function of a standard library class template, or

(b) an explicit specialization of any member function template of a standard library class or class template, or

(c) an explicit or partial specialization of any member class template of a standard library class or class template, or

(d) a deduction guide for any standard library class template.

5 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.

6 Let F denote a standard library function (16.5.5.4), a standard library static member function, or an instantiation of a standard library function template. Unless F is designated an addressable function, the behavior of a C++ program is unspecified (possibly ill-formed) if it explicitly or implicitly attempts to form a pointer to F. [Note: Possible means of forming such pointers include application of the unary & operator (7.6.2.1), &addressof (20.10.11), or a function-to-pointer standard conversion (7.3.3). —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 (16.5.5.5) 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.

8 A translation unit shall not declare namespace std to be an inline namespace (9.8.1).

16.5.4.2.2 Namespace posix

1 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.

16.5.4.2.3 Namespaces for future standardization

1 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: The top-level namespace std2 is reserved for use by future revisions of this International Standard. —end example]

172) 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.

173) 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 may 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.
16.5.4.3  Reserved names

The C++ standard library reserves the following kinds of names:

1. macros
2. global names
3. names with external linkage

If a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this Clause, its behavior is undefined.

16.5.4.3.1 Zombie names

In namespace `std`, the following names are reserved for previous standardization:

1. `auto_ptr`
2. `auto_ptr_ref`
3. `binary_function`
4. `binary_negate`
5. `bindist`
6. `bind2nd`
7. `binder1st`
8. `binder2nd`
9. `const_mem_fun1_ref_t`
10. `const_mem_fun1_t`
11. `const_mem_fun_ref_t`
12. `const_mem_fun_t`
13. `get_temporary_buffer`
14. `get_unexpected`
15. `gets`
16. `is_literal_type`
17. `is_literal_type_v`
18. `mem_fun1_ref_t`
19. `mem_fun1_t`
20. `mem_fun_ref_t`
21. `mem_fun_ref`
22. `mem_fun_t`
23. `mem_fun`
24. `not1`
25. `not2`
26. `pointer_to_binary_function`
27. `pointer_to_unary_function`
28. `ptr_fun`
29. `random_shuffle`
30. `raw_storage_iterator`
31. `result_of`
32. `result_of_t`
33. `return_temporary_buffer`
34. `set_unexpected`
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:

1. **argument_type**
2. **first_argument_type**
3. **io_state**
4. **open_mode**
5. **second_argument_type**
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.

### 16.5.4.3.2 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).

### 16.5.4.3.3 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, 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.
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, or as a name of namespace scope in the global namespace.

### 16.5.4.3.4 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`.

### 16.5.4.3.5 User-defined literal suffixes

1. Literal suffix identifiers (12.8) that do not start with an underscore are reserved for future standardization.

### 16.5.4.4 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.

174) The list of such reserved names includes `errno`, declared or defined in `<cerrno>` (19.4.1).

175) The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>` (17.13.2), and `va_end(va_list)`, declared or defined in `<stdarg>` (17.13.1).

176) 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.
16.5.4.5 Derived classes

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 (11.7.2).

16.5.4.6 Replacement functions

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 (16.3).

A C++ program may provide the definition for any of the following dynamic memory allocation function signatures declared in header `<new>` (6.7.5.4, 17.6.1):

- `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&)`
- `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&)`

The program’s definitions are used instead of the default versions supplied by the implementation (17.6.2). 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.

16.5.4.7 Handler functions

The C++ standard library provides a default version of the following handler function (Clause 17):

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. `set_new_handler`
3. `set_terminate`

See also subclauses 17.6.3, 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:

4. `get_new_handler`
5. `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.

16.5.4.8 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:
For replacement functions (17.6.2), if the installed replacement function does not implement the semantics of the applicable Required behavior: paragraph.

For handler functions (17.6.3.3, 17.9.4.1), if the installed handler function does not implement the semantics of the applicable Required behavior: paragraph.

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 (16.5.3.5, 22.2, 23.3, 25.2, 26.2). Operations on such types can report a failure by throwing an exception unless otherwise specified.

If any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable Required behavior: paragraph.

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.

16.5.4.9 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.

(1.2) 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.

(1.3) 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: 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 (13.10.2.1) and thus is not covered by the previous sentence. — end note] [Note: 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 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]

16.5.4.10 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 16.5.5.10. [Note: 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: This applies even to objects such as mutexes intended for thread synchronization. — end note]

16.5.4.11 Expects paragraph

Violation of any preconditions specified in a function’s Preconditions: element results in undefined behavior.

16.5.4.12 Semantic requirements

A sequence Args of template arguments is said to model a concept C if Args satisfies C (13.5.2) 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 (16.4.1.3) are not modeled at the point of use, the program is ill-formed, no diagnostic required.
16.5.5 Conforming implementations

16.5.5.1 Overview

Subclause 16.5.5 describes the constraints upon, and latitude of, implementations of the C++ standard library.

An implementation’s use of headers is discussed in 16.5.5.2, its use of macros in 16.5.5.3, non-member functions in 16.5.5.4, member functions in 16.5.5.5, data race avoidance in 16.5.5.10, access specifiers in 16.5.5.11, class derivation in 16.5.5.12, and exceptions in 16.5.5.13.

16.5.5.2 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.9) shall include only their corresponding C++ standard library header, as described in 16.5.1.2.

16.5.5.3 Restrictions on macro definitions

The names and global function signatures described in 16.5.1.1 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.

16.5.5.4 Non-member functions

It is unspecified whether any non-member functions in the C++ standard library are defined as inline (9.2.7).

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.177

An implementation shall not declare a non-member function signature with additional default arguments.

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 (6.5.2). [Note: The phrase “unless otherwise specified” applies to cases such as the swappable with requirements (16.5.3.2). The exception for overloaded operators allows argument-dependent lookup in cases like that of ostream_iterator::operator=(23.6.2.2):

*Effects:
  *out_stream << value;
  if (delim != 0)
    *out_stream << delim;
  return *this;
—end note]

16.5.5.5 Member functions

It is unspecified whether any member functions in the C++ standard library are defined as inline (9.2.7).

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: For instance, an implementation may 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]

16.5.5.6 Friend functions

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: In particular, an implementation is not allowed to provide an additional declaration of that function or function template at namespace scope. —end note]

177) A valid C++ program always calls the expected library non-member function. An implementation may also define additional non-member functions that would otherwise not be called by a valid C++ program.

§ 16.5.5.6
Such a friend function or function template declaration is known as a hidden friend, as it is visible neither to ordinary unqualified lookup (6.5.1) nor to qualified lookup (6.5.3). —end note

16.5.5.7 Constexpr functions and constructors

This document explicitly requires that certain standard library functions are constexpr (9.2.5). 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.

16.5.5.8 Requirements for stable algorithms

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).

16.5.5.9 Reentrancy

Except where explicitly specified in this document, it is implementation-defined which functions in the C++ standard library may be recursively reentered.

16.5.5.10 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.

2 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.

3 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: 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]

5 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.

6 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: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. —end note]

7 Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

8 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.

9 [Note: This allows implementations to parallelize operations if there are no visible side effects. —end note]

16.5.5.11 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.

16.5.5.12 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:

1. Every base class described as `virtual` shall be virtual;
2. Every base class not specified as `virtual` shall not be virtual;
3. Unless explicitly stated otherwise, types with distinct names shall be distinct types.

All types specified in the C++ standard library shall be non-final types unless otherwise specified.

### 16.5.5.13 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 except when such a function calls a program-supplied function that throws an exception.

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.

An implementation may strengthen the exception specification for a non-virtual function by adding a non-throwing exception specification.

### 16.5.5.14 Restrictions on storage of pointers

Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type `intptr_t` shall store such values in a traceable pointer location. Libraries that store pointers outside the user’s address space should make it appear that they are stored and retrieved from a traceable pointer location.

### 16.5.5.15 Value of error codes

Certain functions in the C++ standard library report errors via a `std::error_code` 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. 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. For errors that do not originate from the operating system, the implementation may provide enums for the associated values.

### 16.5.5.16 Moved-from state of library types

Objects of types defined in the C++ standard library may be moved from. 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.

---

178) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as `size_t` and `streamoff`.

179) 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.

180) The functions `qsort()` and `bsearch()` meet this condition.

181) 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`.

§ 16.5.5.16
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;, &lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>17.3 Implementation properties</td>
<td><code>&lt;float&gt;, &lt;climits&gt;, &lt;limits&gt;, &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;, &lt;csignal&gt;, &lt;cstdio&gt;, &lt;cstdlib&gt;</code></td>
</tr>
</tbody>
</table>

17.2 Common definitions

17.2.1 Header `<cstdint>` 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;
```

§ 17.2.1
template<class IntType>
    constexpr IntType to_integer(byte b) noexcept;
}
#define NULL see below
#define offsetof(P, D) see below

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`, 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

namespace std {
    using size_t = see below;
    using div_t = see below;
    using ldiv_t = see below;
    using lldiv_t = see below;
}
#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 int atol(const char* nptr);
    long long int 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);
unsigned long long int strtoull(const char* nptr, char** endptr, int base);

// 21.5.6, multibyte / wide string and character conversion functions
int mbstrlen(const char* s, size_t n);
int mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);

// 25.12, C standard library algorithms
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
              c-compare-pred* compar);

void qsort(void* base, size_t nmemb, size_t size,
          c-compare-pred* compar);

// 26.6.9, low-quality random number generation
int rand();
void srand(unsigned int seed);

// 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);

long int labs(long int j);
long long int llabs(long long int j);

div_t div(int numerator, int denominator);
ldiv_t ldiv(long int numerator, long int denominator); // see 16.2
lldiv_t lldiv(long long int numerator, long long int denominator); // see 16.2

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, 26.6.9, and 26.8.2. [Note: 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 [support.types.nullptr]

The type nullptr_t is a synonym for the type of a nullptr expression, and it has the characteristics described in 6.8.1 and 7.3.11. [Note: Although nullptr_t’s address cannot be taken, the address of another nullptr_t object that is an lvalue can be taken. — end note]

The macro NULL is an implementation-defined null pointer constant.182

See also: ISO C 7.19

17.2.4 Sizes, alignments, and offsets [support.types.layout]

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.183

The expression offsetof(type, member-designator) is never type-dependent (13.8.2.2) and it is value-dependent (13.8.2.3) if and only if type is dependent. The result of applying the offsetof macro to a static

182) Possible definitions include 0 and 0L, but not (void*)0.
183) Note that offsetof is required to work as specified even if unary operator& is overloaded for any of the types involved.
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.

2 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.

3 The type size_t is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object (7.6.2.4).

4 [Note: It is recommended that implementations choose types for ptrdiff_t and size_t whose integer conversion ranks (6.8.4) are no greater than that of signed long int unless a larger size is necessary to contain all the possible values. — end note]

5 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 [support.types.byteops]

template<class IntType>
constexpr byte& operator<<=(byte& b, IntType shift) noexcept;

1 Constraints: is_integral_v<IntType> is true.

2 Effects: Equivalent to: return b = b << shift;

template<class IntType>
constexpr byte operator<<(byte b, IntType shift) noexcept;

3 Constraints: is_integral_v<IntType> is true.

4 Effects: Equivalent to:
return static_cast<byte>(static_cast<unsigned int>(b) << shift);

template<class IntType>
constexpr byte& operator>>(byte& b, IntType shift) noexcept;

5 Constraints: is_integral_v<IntType> is true.

6 Effects: Equivalent to: return b = b >> shift;

template<class IntType>
constexpr byte operator>>(byte b, IntType shift) noexcept;

7 Constraints: is_integral_v<IntType> is true.

8 Effects: Equivalent to:
return static_cast<byte>(static_cast<unsigned int>(b) >> shift);

constexpr byte& operator|=(byte& l, byte r) noexcept;

9 Effects: Equivalent to: return l = l | r;

constexpr byte operator|(byte l, byte r) noexcept;

10 Effects: Equivalent to:
return static_cast<byte>(static_cast<unsigned int>(l) | static_cast<unsigned int>(r));

constexpr byte& operator&=(byte& l, byte r) noexcept;

11 Effects: Equivalent to: return l = l & r;

constexpr byte operator&(byte l, byte r) noexcept;

12 Effects: Equivalent to:
return static_cast<byte>(static_cast<unsigned int>(l) & static_cast<unsigned int>(r));

constexpr byte& operator^=(byte& l, byte r) noexcept;

13 Effects: Equivalent to: return l = l ^ r;
constexpr byte operator~(byte b) noexcept;
15
Effects: Equivalent to:

return static_cast<byte>(~static_cast<unsigned int>(b));

17.3 Implementation properties [support.limits]
17.3.1 General [support.limits.general]
1 The headers <limits> (17.3.3), <climits> (17.3.6), and <cfloat> (17.3.7) supply characteristics of imple-
mentation-dependent arithmetic types (6.8.1).
17.3.2 Header <version> synopsis [version.syn]
1 The header <version> supplies implementation-dependent information about the C++ standard library (e.g.,
version number and release date).
2 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: Future versions of this International
Standard might replace the values of these macros with greater values. — end note]
#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_as_const 201510L // also in <utility>
#define __cpp_lib_assume_aligned 201907L // also in <memory>
#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_barrier 201907L // also in <barrier>
#define __cpp_lib_bit_cast 201806L // also in <algorithm>
#define __cpp_lib_bitops 201907L // also in <atomic>, <chrono>
#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 201902L // also in <coroutine>
#define __cpp_lib_destroying_delete 201806L // also in <new>
#define __cpp_lib_enable_shared_from_this 201603L // also in <memory>
#define __cpp_lib_endian 201907L // also in <bit>
#define __cpp_lib_erase_if 202002L // also in <string>, <deque>, <forward_list>, <list>, <vector>, <map>, <set>, <unordered_map>, <unordered_set>
#define __cpp_lib_exchange_function 201304L // also in <utility>
#define __cpp_lib_execution 201902L // also in <execution>
#define __cpp_lib_filesystem 201703L // also in <filesystem>
#define __cpp_lib_format 201907L // also in <format>
#define __cpp_lib_gcd_lcm 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 201703L // also in <new>
#define __cpp_lib_has_unique_object_representations 201606L // also in <type_traits>
#define __cpp_lib_hypot 201603L // also in <cmath>
#define __cpp_lib_hypot2 202002L // also in <bit>
#define __cpp_lib_integer_comparison_functions 202002L // also in <utility>
#define __cpp_lib_integer_sequence 201304L // also in <utility>
#define __cpp_lib_interpolate 201907L // also in <functional>, <numeric>
#define __cpp_lib_invoke 201411L // also in <functional>
#define __cpp_lib_is_aggregate 201703L // also in <type_traits>
#define __cpp_lib_is_aggregate 201811L // 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_incomplete_container_elements 201505L // also in <forward_list>, <list>, <vector>
#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_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_int_pow2 202002L // also in <functional>
#define __cpp_lib_int_pow2 202002L // also in <cmath>
#define __cpp_lib_interpolate 201907L // also in <functional>, <numeric>
#define __cpp_lib_is_aggregate 201703L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_is_aggregate 201703L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_is_aggregate 201703L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_is_aggregate 201703L // also in <type_traits>
#define __cpp_lib_is_layout_compatible 201907L // also in <type_traits>
#define __cpp_lib_is_nothrow_convertible 201806L // also in <type_traits>
#define __cpp_lib_is_swappable 201603L // also in <type_traits>
#define __cpp_lib_parallel_algorithm 201603L // also in <algorithm>, <numeric>
#define __cpp_lib_polymorphic_allocator 201902L // also in <memory>
#define __cpp_lib_quoted_string_io 201304L // also in <iomanip>
#define __cpp_lib_ranges 201911L // also in <algorithm>, <functional>, <iterator>, <memory>, <ranges>
#define __cpp_lib_raw_memory_algorithms 201606L // also in <memory>
#define __cpp_lib_result_of_sFINAE 201210L // also in <functional>, <type_traits>
#define __cpp_lib_robust_nonmodifying_seq_ops 201304L // also in <algorithm>
#define __cpp_lib_sample 201603L // also in <algorithm>
#define __cpp_lib_scoped_lock 201703L // also in <mutex>
#define __cpp_lib_semaphore 201907L // also in <semaphore>
#define __cpp_lib_shared_mutex 201505L // also in <shared_mutex>
#define __cpp_lib_shared_ptr_arrays 201707L // also in <memory>
#define __cpp_lib_shared_ptr_weak_type 201606L // also in <memory>
#define __cpp_lib_shared_timed_mutex 201402L // also in <shared_mutex>
#define __cpp_lib_shift 201806L // also in <algorithm>
#define __cpp_lib_smart_ptr_for_overwrite 202002L // also in <memory>
#define __cpp_lib_source_location 201907L // also in <source_location>
#define __cpp_lib_span 202002L // also in <span>
#define __cpp_lib_ssize 201902L // also in <iterator>
#define __cpp_lib_starts_ends_with 201711L // also in <string>, <string_view>
#define __cpp_lib_string_udls 201304L // also in <string>
#define __cpp_lib_string_view 201803L // also in <string>, <string_view>
#define __cpp_lib_syncbuf 201803L // also in <synostream>
#define __cpp_lib_three_way_comparison 201907L // also in <compare>
#define __cpp_lib_to_address 201711L // also in <memory>
#define __cpp_lib_to_array 201907L // also in <array>
#define __cpp_lib_to_chars 201611L // also in <ucharconv>
#define __cpp_lib_transformation_trait_aliases 201304L // also in <type_traits>
#define __cpp_lib_transparent_operators 201510L // also in <memory>, <functional>
#define __cpp_lib_tuple_element_t 201402L // also in <tuple>
#define __cpp_lib_tuples_by_type 201304L // also in <utility>, <tuple>
#define __cpp_lib_type_identity 201806L // also in <type_traits>
#define __cpp_lib_type_trait_variable_templates 201510L // also in <type_traits>
#define __cpp_lib_uncaught_exceptions 201411L // also in <exception>
#define __cpp_lib_unordered_map_try_emplace 201411L // also in <unordered_map>
#define __cpp_lib_unwrap_ref 201811L // also in <type_traits>
#define __cpp_lib_variant 201606L // also in <variant>
#define __cpp_lib_void_t 201411L // also in <type_traits>

17.3.3 Header <limits> synopsis [limits.syn]

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>;

§ 17.3.3 494
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

1 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(); }  
    }
}
static constexpr int digits = 0;
static constexpr int digits10 = 0;
static constexpr int max_digits10 = 0;
static constexpr bool is_signed = false;
static constexpr bool is_integer = false;
static constexpr bool is_exact = false;
static constexpr int radix = 0;
static constexpr T epsilon() noexcept { return T(); }
static constexpr T round_error() noexcept { return T(); }
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 T infinity() noexcept { return T(); }
static constexpr T quiet_NaN() noexcept { return T(); }
static constexpr T signaling_NaN() noexcept { return T(); }
static constexpr T denorm_min() noexcept { return T(); }
static constexpr bool is_iec559 = false;
static constexpr bool is_bounded = false;
static constexpr bool is_modulo = false;
static constexpr bool traps = false;
static constexpr float_round_style round_style = round_toward_zero;
};

2 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.

3 The default numeric_limits<T> template shall have all members, but with 0 or false values.

4 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.

5 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.

6 Non-arithmetic standard types, such as complex<T> (26.4.2), shall not have specializations.

17.3.5.1 numeric_limits members

Each member function defined in this subclause is signal-safe (17.13.4).

static constexpr T min() noexcept;
2 Minimum finite value.\(^{(184)}\)
3 For floating-point types with subnormal numbers, returns the minimum positive normalized value.
4 Meaningful for all specializations in which is_bounded != false, or is_bounded == false && is_signed == false.

static constexpr T max() noexcept;
5 Maximum finite value.\(^{(185)}\)
6 Meaningful for all specializations in which is_bounded != false.

\(^{(184)}\) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
\(^{(185)}\) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
static constexpr T lowest() noexcept;
A finite value \(x\) such that there is no other finite value \(y\) where \(y < x\).\(^{186}\)
Meaningful for all specializations in which \textit{is\_bounded} \(!=\ false\).

static constexpr int digits;
Number of \textit{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 \textit{radix} digits in the mantissa.\(^{187}\)

static constexpr int digits10;
Number of base 10 digits that can be represented without change.\(^{188}\)
Meaningful for all specializations in which \textit{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;
\textit{true} if the type is signed.
Meaningful for all specializations.

static constexpr bool is\_integer;
\textit{true} if the type is integer.
Meaningful for all specializations.

static constexpr bool is\_exact;
\textit{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).\(^{189}\)
For integer types, specifies the base of the representation.\(^{190}\)
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.\(^{191}\)
Meaningful for all floating-point types.

static constexpr T round\_error() noexcept;
Measure of the maximum rounding error.\(^{192}\)

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.\(^{193}\)

\(^{186}\) \textit{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.
\(^{187}\) Equivalent to \texttt{FLT\_MANT\_DIG}, \texttt{DBL\_MANT\_DIG}, \texttt{LDBL\_MANT\_DIG}.
\(^{188}\) Equivalent to \texttt{FLT\_DIG}, \texttt{DBL\_DIG}, \texttt{LDBL\_DIG}.
\(^{189}\) Equivalent to \texttt{FLT\_RADIX}.
\(^{190}\) Distinguishes types with bases other than 2 (e.g. BCD).
\(^{191}\) Equivalent to \texttt{FLT\_EPSILON}, \texttt{DBL\_EPSILON}, \texttt{LDBL\_EPSILON}.
\(^{192}\) Rounding error is described in LIA-1 Section 5.2.4 and Annex C Rationale Section C.5.2.4 — Rounding and rounding constants.
\(^{193}\) Equivalent to \texttt{FLT\_MIN\_EXP}, \texttt{DBL\_MIN\_EXP}, \texttt{LDBL\_MIN\_EXP}.

§ 17.3.5.1
Meaningful for all floating-point types.

```cpp
static constexpr int min_exponent10;
```

Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.\(^{194}\)

Meaningful for all floating-point types.

```cpp
static constexpr int max_exponent;
```

Maximum positive integer such that \(\text{radix}\) raised to the power one less than that integer is a representable finite floating-point number.\(^{195}\)

Meaningful for all floating-point types.

```cpp
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.\(^{196}\)

Meaningful for all floating-point types.

```cpp
static constexpr bool has_infinity;
```

`true` if the type has a representation for positive infinity.

Meaningful for all floating-point types.

```cpp
    true for all specializations in which is_iec559 != false.
```

```cpp
static constexpr bool has_quiet_NaN;
```

`true` if the type has a representation for a quiet (non-signaling) “Not a Number”.\(^{197}\)

Meaningful for all floating-point types.

```cpp
    true for all specializations in which is_iec559 != false.
```

```cpp
static constexpr bool has_signaling_NaN;
```

`true` if the type has a representation for a signaling “Not a Number”.\(^{198}\)

Meaningful for all floating-point types.

```cpp
    true for all specializations in which is_iec559 != false.
```

```cpp
static constexpr float_denorm_style has_denorm;
```

- `denorm_present` if the type allows subnormal values (variable number of exponent bits)\(^{199}\), `denorm_absent` if the type does not allow subnormal values, and `denorm_indeterminate` if it is indeterminate at compile time whether the type allows subnormal values.

Meaningful for all floating-point types.

```cpp
static constexpr bool has_denorm_loss;
```

`true` if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.\(^{200}\)

```cpp
static constexpr T infinity() noexcept;
```

Representation of positive infinity, if available.\(^{201}\)

Meaningful for all specializations for which `has_infinity` != `false`. Required in specializations for which is_iec559 != `false`.\(^{201}\)

---

\(^{194}\) Equivalent to `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.\(^{195}\) Equivalent to `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.\(^{196}\) Equivalent to `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.\(^{197}\) Required by LIA-1.\(^{198}\) Required by LIA-1.\(^{199}\) Required by LIA-1.\(^{200}\) See ISO/IEC/IEEE 60559.\(^{201}\) Required by LIA-1.
static constexpr T quiet_NaN() noexcept;
    Representation of a quiet “Not a Number”, if available.\(^{202}\)
    Meaningful for all specializations for which has_quiet_NaN != false. Required in specializations for which is_iec559 != false.

static constexpr T signaling_NaN() noexcept;
    Representation of a signaling “Not a Number”, if available.\(^{203}\)
    Meaningful for all specializations for which has_signaling_NaN != false. Required in specializations for which is_iec559 != false.

static constexpr T denorm_min() noexcept;
    Minimum positive subnormal value.\(^{204}\)
    Meaningful for all floating-point types.
    In specializations for which has_denorm == false, returns the minimum positive normalized value.

static constexpr bool is_iec559;
    true if and only if the type adheres to ISO/IEC/IEEE 60559.\(^{205}\)
    Meaningful for all floating-point types.

static constexpr bool is_bounded;
    true if the set of values representable by the type is finite.\(^{206}\) [Note: All fundamental types (6.8.1) are bounded. This member would be false for arbitrary precision types. — end note]
    Meaningful for all specializations.

static constexpr bool is_modulo;
    true if the type is modulo.\(^{207}\) A type is modulo if, for any operation involving +, -, or * on values of that type whose result would fall outside the range \([\min(), \max()]\), the value returned differs from the true value by an integer multiple of \(\max() - \min() + 1\).
    [Example: is_modulo is false for signed integer types (6.8.1) unless an implementation, as an extension to this document, defines signed integer overflow to wrap. — end example]
    Meaningful for all specializations.

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.\(^{208}\)
    Meaningful for all specializations.

static constexpr bool tinyness_before;
    true if tinyness is detected before rounding.\(^{209}\)
    Meaningful for all floating-point types.

static constexpr float_round_style round_style;
    The rounding style for the type.\(^{210}\)
    Meaningful for all floating-point types. Specializations for integer types shall return round_toward_-zero.

\(^{202}\) Required by LIA-1.
\(^{203}\) Required by LIA-1.
\(^{204}\) Required by LIA-1.
\(^{206}\) Required by LIA-1.
\(^{207}\) Required by LIA-1.
\(^{208}\) Required by LIA-1.
\(^{209}\) Required by LIA-1.
\(^{210}\) Equivalent to FLT_ROUNDS. Required by LIA-1.
17.3.5.2 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, epsilon() is only meaningful if is_integer is false). Any value that is not "meaningful" shall be set to 0 or false.

Example:

```cpp
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_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;
    };
}

— end example]

The specialization for bool shall be provided as follows:

```cpp
namespace std {
    template<> class numeric_limits<bool> {
    public:
        static constexpr bool is_specialized = true;
        static constexpr bool min() noexcept { return false; }
        static constexpr bool max() noexcept { return true; }
        static constexpr bool lowest() noexcept { return false; }
    };
```

§ 17.3.5.2
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 radius = 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

see below

1 The header <climits> defines all macros the same as the C standard library header <limits.h>. [Note:
The types of the constants defined by macros in <climits> are not required to match the types to which the
macros refer. —end note]

See also: ISO C 5.2.4.2.1
17.3.7 Header <cfloat> synopsis

```c
#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 DBL_DECIMAL_DIG see below
#define LDBL_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_10_EXP see below
#define DBL_MAX_10_EXP see below
#define LDBL_MAX_10_EXP 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

17.4.1 Header `<cstdint>` synopsis

```c
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;
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

}  // end of inttypes.h

The header also defines numerous macros of the form:

INT{[FAST LEAST]{8 16 32 64}_MIN
INT{[MAX PTR]{8 16 32 64}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX

plus function macros of the form:

{UINT[8 16 32 64}_MAX}_C

The header defines all types and macros the same as the C standard library header <stdint.h>.

See also: ISO C 7.20

17.5  Startup and termination

[support.start.term]

1  [Note: The header <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 atexit() (6.9.3.4). The function
_Exit is signal-safe (17.13.4).

[[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 atexit() (6.9.3.4). The function
abort is signal-safe (17.13.4).

int atexit(c-atexit-handler* f) noexcept;
int atexit(atexit-handler* f) noexcept;

6  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: The atexit() functions do not introduce a data
race (16.5.5.10). — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

7  Returns: The atexit() function returns zero if the registration succeeds, nonzero if it fails.

§ 17.5  503
[[noexcept]] void exit(int status);

**Effects:**
- 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()**.)

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 (14.6.1).

- 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.

- 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.

```c
int at_quick_exit(c-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: The **at_quick_exit()** functions do not introduce a data race (16.5.5.10). — end note] [Note: The order of registration may be indeterminate if **at_quick_exit** was called from more than one thread. — end note] [Note: The **at_quick_exit** registrations are distinct from the **atexit** registrations, and applications may 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.

```c
[[noexcept]] 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: 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 should not 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 (17.13.4) when the functions registered with **at_quick_exit** are.

**See also:** ISO C 7.22.4

### 17.6 Dynamic memory management [support.dynamic]

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.

#### 17.6.1 Header `<new>` synopsis [new.syn]

```cpp
namespace std {
    // 17.6.3, storage allocation errors
    class bad_alloc;
    class bad_array_new_length;
}
```

211) A function is called for every time it is registered.

212) 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**.

213) The macros **EXIT_FAILURE** and **EXIT_SUCCESS** are defined in `<cstdlib>` (17.2.2).
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 nothrow_t { explicit nothrow_t() = default; };
const nothrow_t nothrow;
using new_handler = void(*)();
new_handler get_new_handler() noexcept;
new_handler set_new_handler(new_handler new_p) noexcept;

// 17.6.4, pointer optimization barrier
template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;

// 17.6.5, hardware interference size
inline constexpr size_t hardware_destructive_interference_size = implementation-defined;
inline constexpr size_t hardware_constructive_interference_size = implementation-defined;

// 17.6.2, 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, 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, void* ptr) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, void* ptr) noexcept;
void operator delete (void* ptr, void*) noexcept;
void operator delete[](void* ptr, void*) noexcept;

17.6.2 Storage allocation and deallocation

1 Except where otherwise specified, the provisions of 6.7.5.4 apply to the library versions of operator new and 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.

17.6.2.1 Single-object forms

[[nodiscard]] void* operator new(std::size_t size);
[[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: The deallocation functions (6.7.5.4.2) called by a delete-expression (7.6.2.8) 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: The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. —end note]

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.

§ 17.6.2.1
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, may, however, increase the size argument to operator new[] to obtain space to store supplemental information.
© ISO/IEC N4861

[[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 (6.7.5.4), 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;

Effects: The deallocation functions (6.7.5.4.2) 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: The default behavior below may change in the future, which will require replacing both deallocation functions when replacing the allocation function. —end note]

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[].

Preconditions: If an implementation has strict pointer safety (6.7.5.4.3) then ptr is a safely-derived pointer.

Preconditions: 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.

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: 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. The functions that do not have a size parameter forward their parameters to the corresponding operator delete (single-object) function.

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;

Effects: The deallocation functions (6.7.5.4.2) 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.

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[].

§ 17.6.2.2 508
Preconditions: If an implementation has strict pointer safety (6.7.5.4.3) then \( \text{ptr} \) is a safely-derived pointer.

Preconditions: If the alignment parameter is not present, \( \text{ptr} \) was returned by an allocation function without an alignment parameter. If present, the alignment argument passed to the allocation function that returned \( \text{ptr} \).

Default behavior: Calls \texttt{operator delete[]}\((\text{ptr})\), or \texttt{operator delete[]}\((\text{ptr}, \text{alignment})\), respectively.

17.6.2.3 Non-allocating forms

These functions are reserved; a C++ program may not define functions that displace the versions in the C++ standard library (16.5.4). The provisions of 6.7.5.4 do not apply to these reserved placement forms of \texttt{operator new} and \texttt{operator delete}.

\[
\begin{align*}
\texttt{[[\text{nodiscard}] void* operator new(\text{std::size_t size, void* ptr}) noexcept;}} & \\
\text{Returns:} \ \text{ptr}. & \\
\text{Remarks:} \text{ Intentionally performs no other action.} & \\
\text{Example:} \text{ This can be useful for constructing an object at a known address:} & \\
\texttt{void* place = operator new(sizeof(Something));} & \\
\texttt{Something* p = new (place) Something();} & \\
\text{— end example} & \\
\texttt{[[\text{nodiscard}] void* operator new[](\text{std::size_t size, void* ptr}) noexcept;}} & \\
\text{Returns:} \ \text{ptr}. & \\
\text{Remarks:} \text{ Intentionally performs no other action.} & \\
\texttt{void operator delete(void* ptr, void*) noexcept;}} & \\
\text{Effects:} \text{ Intentionally performs no action.} & \\
\text{Preconditions:} \text{ If an implementation has strict pointer safety (6.7.5.4.3) then \( \text{ptr} \) is a safely-derived pointer.} & \\
\text{Remarks:} \text{ 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 (7.6.2.7).} & \\
\texttt{void operator delete[](void* ptr, void*) noexcept;}} & \\
\text{Effects:} \text{ Intentionally performs no action.} & \\
\text{Preconditions:} \text{ If an implementation has strict pointer safety (6.7.5.4.3) then \( \text{ptr} \) is a safely-derived pointer.} & \\
\text{Remarks:} \text{ 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 (7.6.2.7).} & \\
\end{align*}
\]

17.6.2.4 Data races

For purposes of determining the existence of data races, the library versions of \texttt{operator new}, user replacement versions of global \texttt{operator new}, the C standard library functions \texttt{aligned_alloc}, \texttt{calloc}, and \texttt{malloc}, the library versions of \texttt{operator delete}, user replacement versions of \texttt{operator delete}, the C standard library function \texttt{free}, and the C standard library function \texttt{realloc} shall not introduce a data race (16.5.5.10). 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.

17.6.3 Storage allocation errors

17.6.3.1 Class \texttt{bad_alloc}

```cpp
namespace std {
  class bad_alloc : public exception {
    public:
      // see 17.9.2 for the specification of the special member functions
      const char* what() const noexcept override;
  }
}
```
The class `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 NTBS.

### 17.6.3.2 Class `bad_array_new_length`

```cpp
namespace std {
    class bad_array_new_length : public bad_alloc {
        public:
            // see 17.9.2 for the specification of the special member functions
            const char* what() const noexcept override;
    };
}
```

The class `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 (7.6.2.7).

```cpp
const char* what() const noexcept override;
```

Returns: An implementation-defined NTBS.

### 17.6.3.3 Type `new_handler`

```cpp
using new_handler = void (*)();
```

The type of a handler function to be called by `operator new()` or `operator new[]()` (17.6.2) when they cannot satisfy a request for additional storage.

**Required behavior:** A `new_handler` shall perform one of the following:

- make more storage available for allocation and then return;
- throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;
- terminate execution of the program without returning to the caller.

### 17.6.3.4 `set_new_handler`

```cpp
new_handler set_new_handler(new_handler new_p) noexcept;
```

**Effects:** Establishes the function designated by `new_p` as the current `new_handler`.

**Returns:** The previous `new_handler`.

**Remarks:** The initial `new_handler` is a null pointer.

### 17.6.3.5 `get_new_handler`

```cpp
new_handler get_new_handler() noexcept;
```

**Returns:** The current `new_handler`. [Note: This may be a null pointer value. — end note]

### 17.6.4 Pointer optimization barrier

```cpp
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 (7.3.5) 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: 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]

17.6.5 Hardware interference size

```cpp
inline constexpr size_t hardware_destructive_interference_size = implementation-defined;
```

1 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:
```cpp
struct keep_apart {
    alignas(hardware_destructive_interference_size) atomic<int> cat;
    alignas(hardware_destructive_interference_size) atomic<int> dog;
};
—end example]
```

```cpp
inline constexpr size_t hardware_constructive_interference_size = implementation-defined;
```

2 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:
```cpp
struct together {
    atomic<int> dog;
    int puppy;
};
struct kennel {
    // Other data members...
    alignas(sizeof(together)) together pack;
    // Other data members...
};
static_assert(sizeof(together) <= hardware_constructive_interference_size);
—end example]
```

17.7 Type identification

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.

17.7.1 Header `<typeinfo>` synopsis

```cpp
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```
17.7.2 Class `type_info` [type.info]

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
        
        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 (7.6.1.7). 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.

`bool operator==(const type_info& rhs) const noexcept;`
1 Effects: Compares the current object with `rhs`.
2 Returns: `true` if the two values describe the same type.

`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.

`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.

`const char* name() const noexcept;`
8 Returns: An implementation-defined NTBS.
9 Remarks: The message may be a null-terminated multibyte string (16.4.2.25.2), suitable for conversion and display as a `wstring` (21.3, 28.4.1.4).

17.7.3 Class `bad_cast` [bad.cast]

namespace std {
    class bad_cast : public exception {
        public:
            const char* what() const noexcept override;
    };
}

1 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 (7.6.1.6).

`const char* what() const noexcept override;`
2 Returns: An implementation-defined NTBS.

17.7.4 Class `bad_typeid` [bad.typeid]

namespace std {
    class bad_typeid : public exception {
        public:
            const char* what() const noexcept override;
    };
}

§ 17.7.4
The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a `typeid` expression (7.6.1.7).

```cpp
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.

```cpp
namespace std {
  struct source_location;
}
```

### 17.8.2 Class `source_location`

```cpp
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
  
  }; // struct source_location
}
```

1. The type `source_location` meets the `Cpp17DefaultConstructible`, `Cpp17CopyConstructible`, `Cpp17CopyAssignable`, and `Cpp17Destructible` requirements (16.5.3.1). Lvalues of type `source_location` are swappable (16.5.3.2). All of the following conditions are true:

   - (1.1) `is_nothrow_move_constructible_v<source_location>`
   - (1.2) `is_nothrow_move_assignable_v<source_location>`
   - (1.3) `is_nothrow_swappable_v<source_location>`

   [Note: 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]

2. The data members `file_name_` and `function_name_` always each refer to an NTBS.

3. 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) `strcmp(lhs.file_name(), rhs_p.file_name()) == 0`
   - (3.2) `strcmp(lhs.function_name(), rhs_p.function_name()) == 0`
   - (3.3) `lhs.line() == rhs_p.line()`
   - (3.4) `lhs.column() == rhs_p.column()`
### 17.8.2.1 Creation

**static consteval source_location current() noexcept;**

1. **Returns:**

   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>
<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 <code>ntbs</code>.</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>

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 (9.3.3.6), or as a subexpression thereof, should correspond to the location of the invocation of the function that uses the default argument (7.6.1.2).

3. **Example:**

```cpp
class 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 (9.3.3.6)
    {}
    s(int blather) : other_member(blather) // values of member refer to this location
    {}
    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
}
```
4 constexpr source_location() noexcept;

Effects: The data members are initialized with valid but unspecified values.

17.8.2.2 Observers

constexpr uint_least32_t line() const noexcept;

Returns: line_.

constexpr uint_least32_t column() const noexcept;

Returns: column_.

constexpr const char* file_name() const noexcept;

Returns: file_name_.

constexpr const char* function_name() const noexcept;

Returns: function_name_.

17.9 Exception handling

The header <exception> defines several types and functions related to the handling of exceptions in a C++ program.

17.9.1 Header <exception> synopsis

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;
    template<class T> [[noreturn]] void throw_with_nested(T&& t);
    template<class E> void rethrow_if_nested(const E& e);
}

17.9.2 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):
default constructor (unless the class synopsis shows other constructors)

copy constructor

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 \( \text{strcmp(\text{lhs}.\text{what}(), \text{rhs}.\text{what}())} \) is equal to 0. The \( \text{what()} \) member function of each such \( T \) satisfies the constraints specified for \( \text{exception::what()} \) (see below).

\[
\text{exception(const exception& rhs) noexcept;}
\text{exception& operator=(const exception& rhs) noexcept;}
\]

Postconditions: If \( *\text{this} \) and \( \text{rhs} \) both have dynamic type \( \text{exception} \) then the value of the expression \( \text{strcmp(\text{what}(), \text{rhs}.\text{what}())} \) shall equal 0.

\[
\text{virtual ~exception();}
\]

Effects: Destroys an object of class \( \text{exception} \).

\[
\text{virtual const char* what() const noexcept;}
\]

Returns: An implementation-defined \( \text{nrb} \).

Remarks: The message may be a null-terminated multibyte string (16.4.2.25.2), suitable for conversion and display as a \( \text{wstring} \) (21.3, 28.4.1.4). The return value remains valid until the exception object from which it is obtained is destroyed or a non-\( \text{const} \) member function of the exception object is called.

### 17.9.3 Class bad_exception

namespace std {

class bad_exception : public exception {
    public:
        // see 17.9.2 for the specification of the special member functions
        const char* what() const noexcept override;
    }
}

The class \( \text{bad_exception} \) defines the type of the object referenced by the \( \text{exception_ptr} \) returned from a call to \( \text{current_exception} \) (17.9.6) when the currently active exception object fails to copy.

\[
\text{const char* what() const noexcept override;}
\]

Returns: An implementation-defined \( \text{nrb} \).

### 17.9.4 Abnormal termination

#### 17.9.4.1 Type terminate_handler

using terminate_handler = void (*)();

The type of a \( \text{handler function} \) to be called by \( \text{std::terminate()} \) when terminating exception processing.

Required behavior: A \( \text{terminate_handler} \) shall terminate execution of the program without returning to the caller.

Default behavior: The implementation’s default \( \text{terminate_handler} \) calls \( \text{abort()} \).

#### 17.9.4.2 set_terminate

\[
\text{terminate_handler set_terminate(terminate_handler f) noexcept;}
\]

Effects: Establishes the function designated by \( f \) as the current handler function for terminating exception processing.

Remarks: It is unspecified whether a null pointer value designates the default \( \text{terminate_handler} \).

Returns: The previous \( \text{terminate_handler} \).
17.9.4.3  get_terminate

terminate_handler get_terminate() noexcept;

1  Returns: The current terminate_handler. [Note: This may be a null pointer value. — end note]

17.9.4.4  terminate

[[noreturn]] void terminate() noexcept;

1  Remarks: Called by the implementation when exception handling must be abandoned for any of several reasons (14.6.1). May also be called directly by the program.

2  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: A default terminate_handler is always considered a callable handler in this context. — end note]

17.9.5  uncaught_exceptions

int uncaught_exceptions() noexcept;

1  Returns: The number of uncaught exceptions (14.6.2).

2  Remarks: When uncaught_exceptions() > 0, throwing an exception can result in a call of the function std::terminate (14.6.1).

17.9.6  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.

6  [Note: An implementation might use a reference-counted smart pointer as exception_ptr. — end note]

7  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. [Note: If rethrow_exception rethrows the same exception object (rather than a copy), concurrent access to that rethrown exception object may introduce a data race. Changes in the number of exception_ptr objects that refer to a particular exception do not introduce a data race. — end note]

exception_ptr current_exception() noexcept;

8  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: 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: The copy constructor of the thrown exception may 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);

9  Preconditions: p is not a null pointer.
10. **Throws:** The exception object to which `p` refers.

```
template<class E> exception_ptr make_exception_ptr(E e) noexcept;
```

11. **Effects:** Creates an `exception_ptr` object that refers to a copy of `e`, as if:
```
try {
  throw e;
} catch(...) {
  return current_exception();
}
```

[Note: This function is provided for convenience and efficiency reasons. — end note]

### 17.9.7 `nested_exception`

namespace std {

```cpp
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;
};
```

```cpp
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: `nested_exception` has a virtual destructor to make it a polymorphic class. Its presence can be tested for with `dynamic_cast`. — end note]

1. **nested_exception() noexcept:**

   **Effects:** The constructor calls `current_exception()` and stores the returned value.

   ```cpp
   [[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`.

   ```cpp
   exception_ptr nested_ptr() const noexcept;
   ```

   **Returns:** The stored exception captured by this `nested_exception` object.

2. **template<class T> [[noreturn]] void throw_with_nested(T&& t):**

   **Preconditions:** `U` meets the `Cpp17CopyConstructible` requirements.

3. **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)`.

4. **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:
   ```cpp
   if (auto p = dynamic_cast<const nested_exception*>(addressof(e)))
     p->rethrow_nested();
   ```
17.10 Initializer lists

The header `<initializer_list>` defines a class template and several support functions related to list-initialization (see 9.4.4). All functions specified in this subclause are signal-safe (17.13.4).

17.10.1 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
    };

    // 17.10.4, 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;
};

An object of type `initializer_list<E>` provides access to an array of objects of type `const E`. [Note: 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 9.4.4. 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.

17.10.2 Initializer list constructors

`constexpr initializer_list() noexcept;`

Postconditions: size() == 0.

17.10.3 Initializer list 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.

17.10.4 Initializer list range access

`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

17.11.1 Header <compare> synopsis

The header <compare> specifies types, objects, and functions for use primarily in connection with the three-way comparison operator (7.6.8).

```cpp
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;

  // 20.14.7.7, 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 eq { equal = 0, equivalent = equal, nonequal = 1, nonequivalent = nonequal }; // exposition only
enum class ord { less = -1, greater = 1 }; // exposition only
enum class ncmp { unordered = -127 }; // exposition only
```
Note: 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: `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` [cmp.partialord]

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.

```cpp
namespace std {
    class partial_ordering {
        int value;  // exposition only
        bool is_ordered;  // exposition only

        // exposition-only constructors
        constexpr explicit partial_ordering(eq v) noexcept : value(int(v)), is_ordered(true) {}  // exposition only
        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;

        // comparisons
        friend constexpr bool operator==(partial_ordering v, unspecified) noexcept;
        friend constexpr bool operator< (partial_ordering v, unspecified) noexcept;
        friend constexpr bool operator> (partial_ordering v, unspecified) noexcept;
        friend constexpr bool operator<=(partial_ordering v, unspecified) noexcept;
        friend constexpr bool operator>=(partial_ordering v, unspecified) noexcept;
        friend constexpr bool operator< (unspecified, partial_ordering v) noexcept;
        friend constexpr bool operator> (unspecified, partial_ordering v) noexcept;
        friend constexpr partial_ordering operator<=>(partial_ordering v, unspecified) noexcept;
        friend constexpr partial_ordering operator<=>(unspecified, partial_ordering v) noexcept;
    }

    constexpr bool operator==(partial_ordering v, unspecified) noexcept;
    constexpr bool operator< (partial_ordering v, unspecified) noexcept;
    constexpr bool operator> (partial_ordering v, unspecified) noexcept;
    constexpr bool operator<=(partial_ordering v, unspecified) noexcept;
    constexpr bool operator>=(partial_ordering v, unspecified) noexcept;
}
```

215) That is, `a < b`, `a == b`, and `a > b` might all be false.
The \texttt{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(eq v) noexcept : value(int(v)) {} // exposition only
        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, 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;
    }

    // valid values' definitions
    inline constexpr weak\_ordering weak\_ordering::less(ord::less);
    inline constexpr weak\_ordering weak\_ordering::equivalent(eq::equivalent);
    inline constexpr weak\_ordering weak\_ordering::greater(ord::greater);
}

constexpr operator partial\_ordering() const noexcept;

Returns:
value == 0 ? partial\_ordering::equivalent :
value < 0 ? partial\_ordering::less :
partial\_ordering::greater
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.

```cpp
namespace std {
  class strong_ordering {
    int value; // exposition only

    // exposition-only constructors
    constexpr explicit strong_ordering(eq v) noexcept : value(int(v)) {} // exposition only
    constexpr explicit strong_ordering(ord v) noexcept : value(int(v)) {} // exposition only

    public:
      // valid values
      static const strong_ordering less;
      static const strong_ordering equal;
      static const strong_ordering equivalent;
      static const strong_ordering greater;

      // 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, unspecified) noexcept;
      friend constexpr bool operator<(strong_ordering v, unspecified) noexcept;
      friend constexpr bool operator>>(strong_ordering v, unspecified) noexcept;
      friend constexpr bool operator<<(unspecified, strong_ordering v) noexcept;
      friend constexpr bool operator>>(unspecified, strong_ordering v) noexcept;
      friend constexpr strong_ordering operator<<(unspecified, strong_ordering v) 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(eq::equal);
      inline constexpr strong_ordering strong_ordering::equivalent(eq::equivalent);
      inline constexpr strong_ordering strong_ordering::greater(ord::greater);
  }
```

### § 17.11.2.4 523
constexpr operator partial_ordering() const noexcept;

Returns:
value == 0 ? partial_ordering::equivalent :
value < 0 ? partial_ordering::less :
partial_ordering::greater

constexpr operator weak_ordering() const noexcept;

Returns:
value == 0 ? weak_ordering::equivalent :
value < 0 ? weak_ordering::less :
weak_ordering::greater

constexpr bool operator==(strong_ordering v, unspecified) noexcept;
constexpr bool operator< (strong_ordering v, unspecified) noexcept;
constexpr bool operator> (strong_ordering v, unspecified) noexcept;
constexpr bool operator<=(strong_ordering v, unspecified) noexcept;
constexpr bool operator>=(strong_ordering v, unspecified) noexcept;

Returns: v.value @ 0 for operator@.

constexpr bool operator< (unspecified, strong_ordering v) noexcept;
constexpr bool operator> (unspecified, strong_ordering v) noexcept;
constexpr bool operator<= (unspecified, strong_ordering v) noexcept;
constexpr bool operator>= (unspecified, strong_ordering v) noexcept;

Returns: 0 @ v.value for operator@.

constexpr strong_ordering operator<=>(strong_ordering v, unspecified) noexcept;
constexpr strong_ordering operator<=>(unspecified, strong_ordering v) noexcept;

Returns: v < 0 ? strong_ordering::greater : v > 0 ? strong_ordering::less : v.

### 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. [Note: A comparison category type is stronger than another if they are distinct types and an instance of the former can be converted to an instance of the latter. —end note]

```cpp
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: This is std::strong_ordering if the expansion is empty. —end note]

### 17.11.4 Concept three_way_comparable

```cpp
template<class T, class Cat>
concept compares=as = // exposition only
  same_as<common_comparison_category_t<T, Cat>, Cat>;

template<class T, class U>
concept partially-ordered-with = // exposition only
  requires((const remove_reference_t<T>& t, const remove_reference_t<U>& u) {
    { t < u } -> boolean-testable;
    { t > u } -> boolean-testable;
    { t <= u } -> boolean-testable;
    { t >= u } -> boolean-testable;
    { u < t } -> boolean-testable;
    { u > t } -> boolean-testable;
    { u <= t } -> boolean-testable;
  };
```
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:

- (1.1) t < u, t <= u, t >= u, and u >= t have the same domain.
- (1.2) bool(t < u) == bool(u > t) is true,
- (1.3) bool(u < t) == bool(t > u) is true,
- (1.4) bool(t <= u) == bool(u >= t) is true, and
- (1.5) bool(u <= t) == bool(t >= u) is true.

```cpp
template<class T, class Cat = partial_ordering>
concept three_way_comparable =
    weakly-equality-comparable-with<T, T> &&
    partially-ordered-with<T, T> &&
    requires(const remove_reference_t<T>& a, const remove_reference_t<T>& b) {
{ a <=> b } -> compares-as<Cat>;
};
```

Let a and b be lvalues of type const remove_reference_t<T>. T and Cat model `three_way_comparable<T, Cat>` only if:

- (2.1) (a <=> b == 0) == bool(a == b) is true,
- (2.2) (a <=> b != 0) == bool(a != b) is true,
- (2.3) ((a <=> b) <=> 0) and (0 <=> (b <=> a)) are equal,
- (2.4) (a <=> b < 0) == bool(a < b) is true,
- (2.5) (a <=> b > 0) == bool(a > b) is true,
- (2.6) (a <=> b <= 0) == bool(a <= b) is true,
- (2.7) (a <=> b >= 0) == bool(a >= b) is true, and
- (2.8) if Cat is convertible to strong_ordering, T models `totally_ordered (18.5.4)`.

```cpp
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
17.11.5 Result of three-way comparison

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

The name `strong_order` denotes a customization point object (16.4.2.2.6). Given subexpressions `E` and `F`, the expression `strong_order(E, F)` is expression-equivalent (16.3.11) to the following:

1. If the decayed types of `E` and `F` differ, `strong_order(E, F)` is ill-formed.
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`.
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.
4. Otherwise, `strong_ordering(compare_three_way()(E, F))` if it is a well-formed expression.
5. Otherwise, `weak_ordering(strong_order(E, F))` if it is a well-formed expression.
6. Otherwise, `weak_order(E, F)` is ill-formed. [Note: This case can result in substitution failure when `weak_order(E, F)` appears in the immediate context of a template instantiation. — end note]

The name `weak_order` denotes a customization point object (16.4.2.2.6). Given subexpressions `E` and `F`, the expression `weak_order(E, F)` is expression-equivalent (16.3.11) to the following:

1. If the decayed types of `E` and `F` differ, `weak_order(E, F)` is ill-formed.
2. Otherwise, `weak_ordering(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 `std::weak_order`.
3. Otherwise, if the decayed type `T` of `E` is a floating-point type, yields a value of type `weak_ordering` that is consistent with the ordering observed by `T`'s comparison operators and `strong_order`, and if `numeric_limits<T>::is_iec559` is true, is additionally consistent with the following equivalence classes, ordered from lesser to greater:
   1. together, all negative NaN values;
   2. negative infinity;
   3. each normal negative value;
   4. each subnormal negative value;
   5. together, both zero values;
   6. each subnormal positive value;
   7. each normal positive value;
   8. positive infinity;
   9. together, all positive NaN values.
4. Otherwise, `weak_ordering(compare_three_way()(E, F))` if it is a well-formed expression.
5. Otherwise, `weak_ordering(strong_order(E, F))` if it is a well-formed expression.
6. Otherwise, `weak_order(E, F)` is ill-formed. [Note: This case can result in substitution failure when `std::weak_order(E, F)` appears in the immediate context of a template instantiation. — end note]

The name `partial_order` denotes a customization point object (16.4.2.2.6). Given subexpressions `E` and `F`, the expression `partial_order(E, F)` is expression-equivalent (16.3.11) to the following:

1. If the decayed types of `E` and `F` differ, `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, \( \text{partial\_ordering}(\text{weak\_order}(E, F)) \) if it is a well-formed expression.

— Otherwise, \( \text{partial\_order}(E, F) \) is ill-formed. [Note: 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 (16.4.2.2.6). Given subexpressions \( E \) and \( F \), the expression \( \text{compare\_strong\_order\_fallback}(E, F) \) is expression-equivalent (16.3.11) to:

4.1. If the decayed types of \( E \) and \( F \) differ, \( \text{compare\_strong\_order\_fallback}(E, F) \) is ill-formed.

4.2. Otherwise, \( \text{strong\_order}(E, F) \) if it is a well-formed expression.

4.3. Otherwise, if the expressions \( E = F \) and \( E < F \) are both well-formed and convertible to \( \text{bool} \),

\[
E = F ? \text{strong\_ordering\::\equal} : \\
E < F ? \text{strong\_ordering\::\less} : \\
\text{strong\_ordering\::\greater}
\]

except that \( E \) and \( F \) are evaluated only once.

4.4. Otherwise, \( \text{compare\_strong\_order\_fallback}(E, F) \) is ill-formed.

5. The name \( \text{compare\_weak\_order\_fallback} \) denotes a customization point object (16.4.2.2.6). Given subexpressions \( E \) and \( F \), the expression \( \text{compare\_weak\_order\_fallback}(E, F) \) is expression-equivalent (16.3.11) to:

5.1. If the decayed types of \( E \) and \( F \) differ, \( \text{compare\_weak\_order\_fallback}(E, F) \) is ill-formed.

5.2. Otherwise, \( \text{weak\_order}(E, F) \) if it is a well-formed expression.

5.3. Otherwise, if the expressions \( E = F \) and \( E < F \) are both well-formed and convertible to \( \text{bool} \),

\[
E = F ? \text{weak\_ordering\::\equal} : \\
E < F ? \text{weak\_ordering\::\less} : \\
\text{weak\_ordering\::\greater}
\]

except that \( E \) and \( F \) are evaluated only once.

5.4. Otherwise, \( \text{compare\_weak\_order\_fallback}(E, F) \) is ill-formed.

6. The name \( \text{compare\_partial\_order\_fallback} \) denotes a customization point object (16.4.2.2.6). Given subexpressions \( E \) and \( F \), the expression \( \text{compare\_partial\_order\_fallback}(E, F) \) is expression-equivalent (16.3.11) to:

6.1. If the decayed types of \( E \) and \( F \) differ, \( \text{compare\_partial\_order\_fallback}(E, F) \) is ill-formed.

6.2. Otherwise, \( \text{partial\_order}(E, F) \) if it is a well-formed expression.

6.3. Otherwise, if the expressions \( E = F \) and \( E < F \) are both well-formed and convertible to \( \text{bool} \),

\[
E = F ? \text{partial\_ordering\::\equal} : \\
E < F ? \text{partial\_ordering\::\less} : \\
F < E ? \text{partial\_ordering\::\greater} : \\
\text{partial\_ordering\::\unordered}
\]

except that \( E \) and \( F \) are evaluated only once.

6.4. Otherwise, \( \text{compare\_partial\_order\_fallback}(E, F) \) is ill-formed.

17.12 Coroutines

The header `<coroutine>` defines several types providing compile and run-time support for coroutines in a C++ program.

17.12.1 Header `<coroutine>` synopsis

```cpp
#include <compare>
```

// see 17.11.1
namespace std {

// 17.12.2, coroutine traits
template<class R, class... ArgTypes>
struct coroutine_traits;

// 17.12.3, coroutine handle
template<class Promise = void>
struct coroutine_handle;

// 17.12.3.6, comparison operators
constexpr bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
constexpr strong_ordering operator<=>(coroutine_handle<> x, coroutine_handle<> y) noexcept;

// 17.12.3.7, hash support
template<class T> struct hash;
template<class P> struct hash<coroutine_handle<P>>;

// 17.12.4, no-op coroutines
struct noop_coroutine_promise;
template<> struct coroutine_handle<noop_coroutine_promise>;
using noop_coroutine_handle = coroutine_handle<noop_coroutine_promise>;
nop_coroutine_handle noop_coroutine() noexcept;

// 17.12.5, trivial awaits
struct suspend_never;
struct suspend_always;
}

17.12.2 Coroutine traits

This subclause defines requirements on classes representing coroutine traits, and defines the class template coroutine_traits that meets those requirements.

17.12.2.1 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 (13.10.2), then coroutine_traits<R,ArgTypes...> has the following publicly accessible member:

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.

17.12.3 Class template coroutine_handle

namespace std {

template<>
struct coroutine_handle<void> {

// 17.12.3.1, construct/reset
constexpr coroutine_handle() noexcept;
constexpr coroutine_handle(nullptr_t) noexcept;
coroutine_handle& operator=(nullptr_t) noexcept;

// 17.12.3.2, export/import
constexpr void* address() const noexcept;
static constexpr coroutine_handle from_address(void* addr);

// 17.12.3.3, observers
constexpr explicit operator bool() const noexcept;
bool done() const;

}
// 17.12.3.4, resumption
void operator()() const;
void resume() const;
void destroy() const;

private:
    void* ptr; // exposition only
};

template<class Promise>
struct coroutine_handle : coroutine_handle<> {
    // 17.12.3.1, construct/reset
    using coroutine_handle<>::coroutine_handle;
    static coroutine_handle from_promise(Promise&);
    coroutine_handle& operator=(nullptr_t) noexcept;
    // 17.12.3.2, export/import
    static constexpr coroutine_handle from_address(void* addr);
    // 17.12.3.5, 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.

17.12.3.1 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.

Returns: A coroutine handle h referring to the coroutine.

Postconditions: addressof(h.promise()) == addressof(p).

coroutine_handle& operator=(nullptr_t) noexcept;

Postconditions: address() == nullptr.

Returns: *this.

17.12.3.2 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.

17.12.3.3 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.

17.12.3.4 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: 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: A concurrent resumption of the coroutine may 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).

17.12.3.5 Promise access

Promise& promise() const;

Preconditions: *this refers to a coroutine.

Returns: A reference to the promise of the coroutine.

17.12.3.6 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()) .

17.12.3.7 Hash support

template<class P> struct hash<coroutine_handle<P>>;

The specialization is enabled (20.14.18).

17.12.4 No-op coroutines

17.12.4.1 Class noop_coroutine_promise

The class noop_coroutine_promise defines the promise type for the coroutine referred to by noop_coroutine_handle (17.12.1).

17.12.4.2 Class coroutine_handle<noop_coroutine_promise>

namespace std {
  template<>
    struct coroutine_handle<noop_coroutine_promise> : coroutine_handle{}
{
  // 17.12.4.2.1, observers
    constexpr explicit operator bool() const noexcept;
    constexpr bool done() const noexcept;

§ 17.12.4.2
// 17.12.4.2, resumption
constexpr void operator()() const noexcept;
constexpr void resume() const noexcept;
constexpr void destroy() const noexcept;

// 17.12.4.3, promise access
noop_coroutine_promise& promise() const noexcept;

// 17.12.4.4, address
constexpr void* address() const noexcept;

private:
coroutine_handle(unspecified);
};

17.12.4.2.1 Observers [coroutine.handle.noop.observers]
constexpr explicit operator bool() const noexcept;

Returns: true.

constexpr bool done() const noexcept;

Returns: false.

17.12.4.2.2 Resumption [coroutine.handle.noop.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.

17.12.4.2.3 Promise access [coroutine.handle.noop.promise]
nop_coroutine_promise& promise() const noexcept;

Returns: A reference to the promise object associated with this coroutine handle.

17.12.4.2.4 Address [coroutine.handle.noop.address]
constexpr void* address() const noexcept;

Returns: ptr.

Remarks: A noop_coroutine_handle’s ptr is always a non-null pointer value.

17.12.4.3 Function noop_coroutine [coroutine.noop.coroutine]
nop_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.

17.12.5 Trivial awaitables [coroutine.trivial.awaitables]
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: The types `suspend_never` and `suspend_always` can be used to indicate that an `await-expression` should either never suspend or always suspend, and in either case not produce a value. —end note

17.13 Other runtime support  [support.runtime]

1 Headers `<csetjmp>` (nonlocal jumps), `<csignal>` (signal handling), `<cstdarg>` (variable arguments), and `<cstdlib>` (runtime environment `getenv`, `system`), provide further compatibility with C code.

2 Calls to the function `getenv` (17.2.2) shall not introduce a data race (16.5.5.10) provided that nothing modifies the environment. [Note: Calls to the POSIX functions `setenv` and `putenv` modify the environment. —end note]

3 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.

17.13.1 Header `<cstdarg>` synopsis  [cstdarg.syn]

```cpp
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
```

1 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 `...`).\(^{216}\) If the parameter `parmN` is a pack expansion (13.7.3) or an entity resulting from a lambda capture (7.5.5), the program is ill-formed, no diagnostic required. If the parameter `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

17.13.2 Header `<csetjmp>` synopsis  [csetjmp.syn]

```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 `<csetjmp>` are the same as the C standard library header `<setjmp.h>`.

2 The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this document. A `setjmp/longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any objects with automatic storage duration. A call to `setjmp` or `longjmp` has undefined behavior if invoked in a suspension context of a coroutine (7.6.2.3).

See also: ISO C 7.13

17.13.3 Header `<csignal>` synopsis  [csignal.syn]

```cpp
namespace std {
  using sig_atomic_t = see below;

  // 17.13.4, signal handlers
  extern "C" using signal-handler = void(int); // exposition only
  signal-handler* signal(int sig, signal-handler* func);
```

\(^{216}\) Note that `va_start` is required to work as specified even if unary operator\& is overloaded for the type of `parmN`. 

§ 17.13.3
17.13.4 Signal handlers

1 A call to the function `signal` synchronizes with any resulting invocation of the signal handler so installed.

2 A plain lock-free atomic operation is an invocation of a function f from Clause 31, such that:

(2.1) — f is the function `atomic_is_lock_free()`, or

(2.2) — f is the member function `is_lock_free()`, or

(2.3) — f is a non-static member function invoked on an object A, such that A.`is_lock_free()` yields `true`, or

(2.4) — f is a non-member function, and for every pointer-to-atomic argument A passed to f, `atomic_is_lock_free(A)` yields `true`.

3 An evaluation is 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; [Note: 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)\(^{217}\); 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.

4 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

---

\(^{217}\) 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: 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 by this document 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: 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 this document 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. This document uses 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-unqualified 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.

7 [Example:

```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:

(7.1) — Expression #1 does not modify either of its operands, #2 modifies both of its operands, and #3 modifies only its first operand a.

(7.2) — Expression #1 implicitly requires additional expression variations that meet the requirements for c == d (including non-modification), as if the expressions

```cpp
c == b;
c == std::move(b);
c == std::move(d);
c == std::move(b);
```

were declared as well.

(7.3) — 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: 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;
}
```

§ 18.3 535
// 18.4.5, concept common_reference_with
template<class T, class U>
    concept common_reference_with = see below;

// 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;

// 18.6. object concepts
template<class T>  // exposition only
  concept movable = see below;
template<class T>  // exposition only
  concept copyable = see below;
template<class T>  // exposition only
  concept semiregular = see below;
template<class T>  // exposition only
  concept regular = see below;

// 18.7. callable concepts
// 18.7.2. concept invocable
template<class F, class... Args>  // exposition only
  concept invocable = see below;

// 18.7.3. concept regular_invocable
template<class F, class... Args>  // exposition only
  concept regular_invocable = see below;

// 18.7.4. concept predicate
template<class F, class... Args>  // exposition only
  concept predicate = see below;

// 18.7.5. concept relation
template<class R, class T, class U>  // exposition only
  concept relation = see below;

// 18.7.6. concept equivalence_relation
template<class R, class T, class U>  // exposition only
  concept equivalence_relation = see below;

// 18.7.7. concept strict_weak_order
template<class R, class T, class U>  // exposition only
  concept strict_weak_order = see below;

18.4 Language-related concepts [concepts.lang]
18.4.1 General [concepts.lang.general]

1 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 [concept.same]

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: same_as<T, U> subsumes same_as<U, T> and vice versa. — end note]

18.4.3 Concept derived_from [concept.derived]

template<class Derived, class Base>
  concept derived_from =
    is_base_of_v<Derived, Base> &&
    is_convertible_v<Derived*, const volatile Base*>;

[Note: 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 (16.5.5.16).
   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: C could be the same as T, or U, or it could be a different type. C may 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.
3. [Note: Users can customize the behavior of common_reference_with by specializing the basic_common_reference class template (20.15.7.6). — 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: 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>());
    } &&
```

§ 18.4.6 538
Let $C$ be $\text{common_type_t<T, U>}$. Let $t_1$ and $t_2$ be equality-preserving expressions (18.2) such that $\text{decltype((t1))}$ and $\text{decltype((t2))}$ are each $T$, and let $u_1$ and $u_2$ be equality-preserving expressions such that $\text{decltype((u1))}$ and $\text{decltype((u2))}$ are each $U$. $T$ and $U$ model $\text{common_with<T, U>}$ only if:

1. $C(t_1)$ equals $C(t_2)$ if and only if $t_1$ equals $t_2$, and
2. $C(u_1)$ equals $C(u_2)$ if and only if $u_1$ equals $u_2$.

[Note: Users can customize the behavior of $\text{common_with}$ by specializing the $\text{common_type}$ class template (20.15.7.6). —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: $\text{signed_integral}$ can be modeled even by types that are not signed integer types (6.8.1); for example, char. —end note]

[Note: $\text{unsigned_integral}$ can be modeled even by types that are not unsigned integer types (6.8.1); 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> &&
  $\text{common_reference_with<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&>}$ &&
  requires(LHS lhs, RHS&& rhs) {
    (lhs = std::forward<RHS>(rhs)) -> same_as<LHS>;
  };
```

Let:

1. $\text{lhs}$ be an lvalue that refers to an object $\text{lcopy}$ such that $\text{decltype((lhs))}$ is $LHS$,
2. $\text{rhs}$ be an expression such that $\text{decltype((rhs))}$ is $RHS$, and
3. $\text{rcopy}$ be a distinct object that is equal to $\text{rhs}$.

$LHS$ and $RHS$ model $\text{assignable_from<LHS, RHS>}$ only if

1. $\text{addressof(lhs = rhs) == addressof(lcopy)}$.
2. After evaluating $\text{lhs = rhs}$:
   1. $\text{lhs}$ is equal to $\text{rcopy}$, unless $\text{rhs}$ is a non-const xvalue that refers to $\text{lcopy}$.
   2. If $\text{rhs}$ is a non-const xvalue, the resulting state of the object to which it refers is valid but unspecified (16.5.5.16).
   3. Otherwise, if $\text{rhs}$ is a glvalue, the object to which it refers is not modified.

[Note: Assignment need not be a total function (16.4.1.3); 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: \( t_1 \) 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 \( \text{common\_reference\_with}\langle\text{decltype}((t_1)), \text{decltype}((u_1))\rangle \) 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 \( \text{common\_reference\_t}\langle\text{decltype}((t_1)), \text{decltype}((u_1))\rangle \).

The name \texttt{ranges::swap} denotes a customization point object (16.4.2.2.6). The expression \texttt{ranges::swap(E1, E2)} for subexpressions \( E_1 \) and \( E_2 \) is expression-equivalent to an expression \( S \) determined as follows:

1. \( S \) is \((\text{void})\text{swap}(E_1, E_2)\) if \( E_1 \) or \( E_2 \) has class or enumeration type (6.8.2) and that expression is valid, with overload resolution performed in a context that includes the declaration

\[
\begin{align*}
\text{template}<\text{class } T> &\quad \text{void swap}(T\& a, T\& b) = \text{delete};
\end{align*}
\]

and does not include a declaration of \texttt{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 (6.8.2) with equal extent and \texttt{ranges::swap(*E1, *E2)} is a valid expression, \( S \) is \((\text{void})\text{ranges::swap\_ranges}(E_1, E_2)\), except that \( \text{noexcept}(S) \) is equal to \( \text{noexcept(\text{ranges::swap(*E1, *E2)})} \).

3. Otherwise, if \( E_1 \) and \( E_2 \) are lvalues of the same type \( T \) that models \texttt{move\_constructible<T>} and \texttt{assignable\_from<T, T>} \( S \) is an expression that exchanges the denoted values. \( S \) is a constant expression if

\[
\begin{align*}
(2.3.1) &\quad T \text{ is a literal type (6.8),} \\
(2.3.2) &\quad \text{both } E_1 = \text{std::move}(E_2) \text{ and } E_2 = \text{std::move}(E_1) \text{ are constant subexpressions (16.3.6), and} \\
(2.3.3) &\quad \text{the full-expressions of the initializers in the declarations}
\begin{align*}
&T_1(\text{std::move}(E_1)); \\
&T_2(\text{std::move}(E_2));
\end{align*}
\text{are constant subexpressions.}
\end{align*}
\]

\( \text{noexcept}(S) \) is equal to \( \text{is\_nothrow\_move\_constructible\_v<T>} \&\& \text{is\_nothrow\_move\_assignable\_v<T>} \).

4. Otherwise, \texttt{ranges::swap(E1, E2)} is ill-formed. [Note: This case can result in substitution failure when \texttt{ranges::swap(E1, E2)} appears in the immediate context of a template instantiation. — end note]

\[\text{Note: Whenever } \texttt{ranges::swap(E1, E2)} \text{ is a valid expression, it exchanges the values denoted by } E_1 \text{ and } E_2 \text{ and has type } \text{void. — end note}\]

\[
\begin{align*}
\text{template}<\text{class } T> &\quad \text{concept swappable} = \text{requires}(T\& a, T\& b) \{ \text{ranges::swap}(a, b); \};
\end{align*}
\]

\[
\begin{align*}
\text{template}<\text{class } T, \text{class } U> &\quad \text{concept swappable\_with} = \\
&\quad \text{common\_reference\_with\langle T, U \rangle} \&\& \\
&\quad \text{requires}(T\&\& t, U\&\& u) \{ \\
&\quad \text{ranges::swap}(\text{std::forward<T>}(t), \text{std::forward<T>}(t)); \\
&\quad \text{ranges::swap}(\text{std::forward<U>}(u), \text{std::forward<U>}(u)); \\
&\quad \text{ranges::swap}(\text{std::forward<T>}(t), \text{std::forward<U>}(u)); \\
&\quad \text{ranges::swap}(\text{std::forward<U>}(u), \text{std::forward<T>}(t)); \\
&\quad \};
\end{align*}
\]

\[\text{Note: The semantics of the } \texttt{swappable} \text{ and } \texttt{swappable\_with} \text{ concepts are fully defined by the } \texttt{ranges::swap} \text{ customization point. — end note}\]

\[\text{218) The name } \texttt{swap} \text{ is used here unqualified.}\]
5 [Example: 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) {}
        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);
}

— end example]

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: 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

541
template<class T>
concept default_initializable = constructible_from<T> &&
requires { T{}; } &&
is-default-initializable<T>;

For a type T, is-default-initializable<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.

18.4.13 Concept move_constructible [concept.moveconstructible]

template<class T>
concept move_constructible = constructible_from<T, T> && convertible_to<T, T>;

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 move_constructible only if

(1.1) After the definition T u = rv;, u is equal to u2.
(1.2) T(rv) is equal to u2.
(1.3) If T is not const, rv’s resulting state is valid but unspecified (16.5.5.16); otherwise, it is unchanged.

18.4.14 Concept copy_constructible [concept.copyconstructible]

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>;

If T is an object type, then let v be an lvalue of type (possibly const) T or an rvalue of type const T.
T models copy_constructible only if

(1.1) After the definition T u = v;, u is equal to v (18.2) and v is not modified.
(1.2) T(v) is equal to v and does not modify v.

18.5 Comparison concepts [concepts.compare]

18.5.1 General [concepts.compare.general]

Subclause 18.5 describes concepts that establish relationships and orderings on values of possibly differing
object types.

18.5.2 Boolean testability [concept.boolean testifyable]

The exposition-only boolean-testable concept specifies the requirements on expressions that are convertible
to bool and for which the logical operators (7.6.14, 7.6.15, 7.6.2.1) have the conventional semantics.

template<class T>
concept boolean-testable-impl = convertible_to<T, bool>; // exposition only

Let e be an expression such that decltype((e)) is T. T models boolean-testable-impl only if:

(2.1) either remove_cvref_t<T> is not a class type, or name lookup for the names operator&& and
operator|| within the scope of remove_cvref_t<T> as if by class member access lookup (11.8)
results in an empty declaration set; and

(2.2) name lookup for the names operator&& and operator|| in the associated namespaces and entities of
T (6.5.2) finds no disqualifying declaration (defined below).

A disqualifying parameter is a function parameter whose declared type P

(3.1) is not dependent on a template parameter, and there exists an implicit conversion sequence (12.4.3.1)
from e to P; or
A key parameter of a function template \( D \) is a function parameter of type \( cvX \) 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. \[\text{Example: In}\]

```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 \( Z::\text{operator&&} \) contains one key parameter, \( C<T> \ x \), and the declaration of \( Z::\text{operator||} \) contains no key parameters. \[\text{--- end example}\]

A disqualifying declaration is

1. a (non-template) function declaration that contains at least one disqualifying parameter; or
2. a function template declaration that contains at least one disqualifying parameter, where
   1. at least one disqualifying parameter is a key parameter; or
   2. the declaration contains no key parameters; or
   3. the declaration declares a function template that is not visible in its namespace (9.8.1.2).

\[\text{Example: The types } bool, \text{true_type } (20.15.2), \text{int}* \text{, and } \text{bitset<N>::reference } (20.9.2) \text{ model boolean-testable}. \[\text{--- end example}\]

18.5.3 Concept equality_comparable \[\text{concept.equalitycomparable}\]

```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 } -> boolean-testable;
        { t != u } -> boolean-testable;
        { u == t } -> boolean-testable;
        { u != t } -> boolean-testable;
    };
```
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-equality-comparable-with \( T, U \) only if

1. \( t == u, u == t, t != u, \) and \( u != t \) have the same domain.
2. \( \text{bool}(u == t) == \text{bool}(t == u) \).
3. \( \text{bool}(t != u) == !\text{bool}(t == u) \).
4. \( \text{bool}(u != t) == \text{bool}(t != u) \).
template<class T>
  concept equality_comparable = weakly-equality-comparable-with<T, T>;

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: 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>;

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>;

Given a type T, let a, b, and c be lvalues of type const remove_reference_t<T>. T models totally_ordered only if

\[\begin{align*}
(1.1) & \text{ Exactly one of bool}(a < b), \text{ bool}(a > b), \text{ or bool}(a == b) \text{ is true.} \\
(1.2) & \text{ If bool}(a < b) \text{ and bool}(b < c), \text{ then bool}(a < c).} \\
(1.3) & \text{ bool}(a <= b) == \lnot \text{ bool}(b < a).} \\
(1.4) & \text{ bool}(a >= b) == \lnot \text{ bool}(a < b).} \\
\end{align*}\]

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>;

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

\[\begin{align*}
(2.1) & \text{ bool}(t < u) == \text{ bool}(C(t) < C(u)).} \\
(2.2) & \text{ bool}(t > u) == \text{ bool}(C(t) > C(u)).} \\
(2.3) & \text{ bool}(t <= u) == \text{ bool}(C(t) <= C(u)).} \\
(2.4) & \text{ bool}(t >= u) == \text{ bool}(C(t) >= C(u)).} \\
(2.5) & \text{ bool}(u < t) == \text{ bool}(C(u) < C(t)).} \\
(2.6) & \text{ bool}(u > t) == \text{ bool}(C(u) > C(t)).} \\
(2.7) & \text{ bool}(u <= t) == \text{ bool}(C(u) <= C(t)).} \\
(2.8) & \text{ bool}(u >= t) == \text{ bool}(C(u) >= C(t)).} \\
\end{align*}\]
18.6 Object concepts

1 This subclause describes concepts that specify the basis of the value-oriented programming style on which the library is based.

```cpp
template<class T>
concept movable = is_object_v<T> && move_constructible<T> &&
assignable_from<T&, T> && swappable<T>;

template<class T>
concept copyable = copy_constructible<T> && movable<T> && assignable_from<T&, T> &&
assignable_from<T&, const T&> && assignable_from<T&, const T>;

template<class T>
concept semiregular = copyable<T> && default_initializable<T>;

template<class T>
concept regular = semiregular<T> && equality_comparable<T>;
```

2 [Note: The `semiregular` concept is modeled by types that behave similarly to built-in types like `int`, except that they might not be comparable with `==`. — end note]

3 [Note: The `regular` concept is modeled by types that behave similarly to built-in types like `int` and that are comparable with `==`. — end note]

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` [concept.invocable]

1 The `invocable` concept specifies a relationship between a callable type (20.14.2) `F` and a set of argument types `Args...` which can be evaluated by the library function `invoke` (20.14.4).

```cpp
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
};
```

2 [Example: A function that generates random numbers can model `invocable`, since the `invoke` function call expression is not required to be equality-preserving (18.2). — end example]

18.7.3 Concept `regular_invocable` [concept.regularinvocable]

```cpp
template<class F, class... Args>
concept regular_invocable = invocable<F, Args...>;
```

1 The `invoke` function call expression shall be equality-preserving (18.2) and shall not modify the function object or the arguments. [Note: This requirement supersedes the annotation in the definition of `invocable`. — end note]

2 [Example: A random number generator does not model `regular_invocable`. — end example]

3 [Note: The distinction between `invocable` and `regular_invocable` is purely semantic. — end note]

18.7.4 Concept `predicate` [concept.predicate]

```cpp
template<class F, class... Args>
concept predicate =
    regular_invocable<F, Args...> && boolean-testable<invoke_result_t<F, Args...>>;
```

18.7.5 Concept `relation` [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

```
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

```
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:

1.2.1 `comp(a, b) && comp(b, c)` implies `comp(a, c)`
1.2.2 `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)`

[Note: Under these conditions, it can be shown that]

1.3.1 `equiv` is an equivalence relation,
1.3.2 `comp` induces a well-defined relation on the equivalence classes determined by `equiv`, and
1.3.3 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

The C++ standard library provides classes to be used to report certain errors (16.5.5.13) 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.

19.2.1 Header <stdexcept> synopsis

```cpp
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;
}
```

19.2.2 Class logic_error

```cpp
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.

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.

19.2.3 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.

19.2.4 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.

19.2.5 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.

19.2.6 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);
    }
}

§ 19.2.6
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);
Postconditions: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
out_of_range(const char* what_arg);
Postconditions: strcmp(what(), what_arg) == 0.
```

### 19.2.7 Class `runtime_error`

```cpp
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);
Postconditions: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
runtime_error(const char* what_arg);
Postconditions: strcmp(what(), what_arg) == 0.
```

### 19.2.8 Class `range_error`

```cpp
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);
Postconditions: strcmp(what(), what_arg.c_str()) == 0.
```

```cpp
range_error(const char* what_arg);
Postconditions: strcmp(what(), what_arg) == 0.
```

### 19.2.9 Class `overflow_error`

```cpp
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);
Postconditions: strcmp(what(), what_arg.c_str()) == 0.
```
overflow_error(const char* what_arg);

Postconditions: strcmp(what(), what_arg) == 0.

19.2.10 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

The header <cassert> provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

19.3.2 The assert macro

An expression assert(E) is a constant subexpression (16.3.6), 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

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: The intent is to remain in close alignment with the POSIX standard. — end note] A separate errno value shall be provided for each thread.

19.4.1 Header <cerrno> synopsis

#define errno see below
#define EDEADLK see below
#define EDESTADDRREQ see below
#define EDOM see below
#define EXIST see below
#define EFAULT see below
#define EFBIG see below
#define EHOSTUNREACH see below
#define EIDRM 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 EORANGE 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 EROFS see below
#define ESPIPE see below
#define ETIME see below
#define ETIMEDOUT see below
#define EWOULDBLOCK see below
#define EXDEV see below

§ 19.4.1 551
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

1 This subclause 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 this subclause shall not change the value of `errno` (19.4). Implementations should leave the error states provided by other libraries unchanged.

19.5.1 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,            // ENOEXEC
        file_exists,                        // EEXIST
        file_too_large,                     // E2BIG
        filename_too_long,                  // ENAMETOOLONG
        function_not_supported,             // ENSYS
        host_unreachable,                   // ENHOSTUNREACH
        identifier_removed,                 // EIDRM
        illegal_byte_sequence,              // EILSEQ
        inappropriate_io_control_operation, // ENOTTY
        interrupted,                       // EINTR
        invalid_argument,                   // EINVAL
        invalid_seek,                       // ESPPIPE
        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, // ENOTDIR
no_such_process, // ENOTCONN
not_a_directory, // ENOTDIR
not_a_socket, // ENOTSOCK
not_a_stream, // ENOSTR
not_connected, // ENOCONNECT
not_enough_memory, // ENOMEM
not_supported, // ENOTSup
operation_canceled, // ECANCELED
operation_in_progress, // EINPROGRESS
operation_not_permitted, // EPERM
operation_not_supported, // EOPNOTSUPP
operation_would_block, // EWOULDBLOCK
owner_dead, // EOWNERDEAD
permission_denied, // EACCES
protocol_error, // EPROTO
protocol_not_supported, // EPROTONOSUPPORT
read_only_file_system, // EROFS
resource_deadlock_would_occur, // EDEADLK
resource_unavailable_try_again, // EAGAIN
result_out_of_range, // ERANGE
state_not_recoverable, // ENOTRECOVERABLE
stream_timeout, // ETIME
text_file_busy, // ETXTBSY
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 {};

// 19.5.3.5, 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);

// 19.5.4.5, non-member functions
error_condition make_error_condition(errc e) noexcept;

// 19.5.5, comparison functions
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
bool operator==(const error_code& lhs, const error_condition& 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;
// 19.5.6, 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.

19.5.2 Class error_category

19.5.2.1 Overview

1 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 19.5.2.

[Note: error_category objects are passed by reference, and two such objects are equal if they have the same address. This means that applications using custom error_category types should create a single object of each such type. — 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;
}

19.5.2.2 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).

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.

§ 19.5.2.2
virtual string message(int ev) const = 0;

Returns: A string that describes the error condition denoted by ev.

19.5.2.3 Non-virtual members

bool operator==(const error_category& rhs) const noexcept;

Returns: this == &rhs.

strong_ordering operator<=>(const error_category& rhs) const noexcept;

Returns: compare_three_way()(this, &rhs).

[Note: compare_three_way (20.14.7.7) provides a total ordering for pointers. — end note]

19.5.2.4 Program-defined classes derived from error_category

virtual const char* name() const noexcept = 0;

Returns: A string naming the error category.

virtual error_condition default_error_condition(int ev) const noexcept;

Returns: An object of type error_condition that corresponds to ev.

virtual bool equivalent(int code, const error_condition& condition) const noexcept;

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;

Returns: true if, for the category of error represented by *this, code is considered equivalent to condition; otherwise, false.

19.5.2.5 Error category objects

const error_category& generic_category() noexcept;

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;

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: The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX errno value. Thus implementations are given latitude in determining correspondence. — end note]

19.5.3 Class error_code

19.5.3.1 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: Class error_code is an adjunct to error reporting by exception. — end note]
namespace std {
    class error_code {
    public:
        // 19.5.3.2, constructors
        error_code() noexcept;
        error_code(int val, const error_category& cat) noexcept;
        template<class ErrorCodeEnum>
            error_code(ErrorCodeEnum e) noexcept;

        // 19.5.3.3, modifiers
        void assign(int val, const error_category& cat) noexcept;
        template<class ErrorCodeEnum>
            error_code& operator=(ErrorCodeEnum e) noexcept;
        void clear() noexcept;

        // 19.5.3.4, 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
    
    // 19.5.3.5, 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);
    
    19.5.3.2  Constructors

    error_code() noexcept;
    // Postconditions: val_ == 0 and cat_ == &system_category().

    error_code(int val, const error_category& cat) noexcept;
    // Postconditions: val_ == val and cat_ == &cat.

    template<class ErrorCodeEnum>
        error_code(ErrorCodeEnum e) noexcept;
    // Constraints: is_error_code_enum_v<ErrorCodeEnum> is true.
    // Postconditions: *this == make_error_code(e).

    19.5.3.3  Modifiers

    void assign(int val, const error_category& cat) noexcept;
    // Postconditions: val_ == val and cat_ == &cat.

    template<class ErrorCodeEnum>
        error_code& operator=(ErrorCodeEnum e) noexcept;
    // Constraints: is_error_code_enum_v<ErrorCodeEnum> is true.
    // Postconditions: *this == make_error_code(e).
    // Returns: *this.

    void clear() noexcept;
    // Postconditions: value() == 0 and category() == system_category().
}
19.5.3.4 Observers

int value() const noexcept;
1
   Returns: val_.

const error_category& category() const noexcept;
2
   Returns: *cat_.

error_condition default_error_condition() const noexcept;
3
   Returns: category().default_error_condition(value()).

string message() const;
4
   Returns: category().message(value()).

explicit operator bool() const noexcept;
5
   Returns: value() != 0.

19.5.3.5 Non-member functions

error_code make_error_code(errc e) noexcept;
1
   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);
2
   Effects: Equivalent to: return os << ec.category().name() << ':' << ec.value();

19.5.4 Class error_condition

19.5.4.1 Overview

The class error_condition describes an object used to hold values identifying error conditions. [Note: error_condition values are portable abstractions, while error_code values (19.5.3) are implementation specific. —end note]

namespace std {
   class error_condition {
      public:
         // 19.5.4.2, constructors
         error_condition() noexcept;
         error_condition(int val, const error_category& cat) noexcept;
         template<class ErrorConditionEnum>
            error_condition(ErrorConditionEnum e) noexcept;

         // 19.5.4.3, modifiers
         void assign(int val, const error_category& cat) noexcept;
         template<class ErrorConditionEnum>
            error_condition& operator=(ErrorConditionEnum e) noexcept;
         void clear() noexcept;

         // 19.5.4.4, observers
         int value() const noexcept;
         const error_category& category() const noexcept;
         string message() const;
         explicit operator bool() const noexcept;

         private:
             int val_;       // exposition only
             const error_category* cat_; // exposition only
   };
}
19.5.4.2 Constructors

```cpp
error_condition() noexcept;
Postconditions: val_ == 0 and cat_ == &generic_category().
```

```cpp
error_condition(int val, const error_category& cat) noexcept;
Postconditions: val_ == val and cat_ == &cat.
```

```cpp
template<class ErrorConditionEnum>
error_condition(ErrorConditionEnum e) noexcept;
Constraints: is_error_condition_enum_v<ErrorConditionEnum> is true.
Postconditions: *this == make_error_condition(e).
```

19.5.4.3 Modifiers

```cpp
void assign(int val, const error_category& cat) noexcept;
Postconditions: val_ == val and cat_ == &cat.
```

```cpp
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.
```

```cpp
void clear() noexcept;
Postconditions: value() == 0 and category() == generic_category().
```

19.5.4.4 Observers

```cpp
int value() const noexcept;
Returns: val_.
```

```cpp
const error_category& category() const noexcept;
Returns: *cat_.
```

```cpp
string message() const;
Returns: category().message(value()).
```

```cpp
explicit operator bool() const noexcept;
Returns: value() != 0.
```

19.5.4.5 Non-member functions

```cpp
error_condition make_error_condition(errc e) noexcept;
Returns: error_condition(static_cast<int>(e), generic_category()).
```

19.5.5 Comparison functions

```cpp
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
Returns: lhs.category() == rhs.category() && lhs.value() == rhs.value()
```

```cpp
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();

19.5.6 System error hash support

[syserr.hash]

template<> struct hash<error_code>;
template<> struct hash<error_condition>;

The specializations are enabled (20.14.18).

19.5.7 Class system_error

[syserr.syserr]

19.5.7.1 Overview

[syserr.syserr.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.

[Note: If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type bad_alloc (17.6.3.1) 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);
    system_error code() const noexcept;
    const char* what() const noexcept override;
};
}

19.5.7.2 Members

[syserr.syserr.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);

4 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);

5     Postconditions: code() == error_code(ev, ecat) and
         string_view(what()).find(what_arg) != string_view::npos.

system_error(int ev, const error_category& ecat);

6     Postconditions: code() == error_code(ev, ecat).

const error_code& code() const noexcept;

7     Returns: ec or error_code(ev, ecat), from the constructor, as appropriate.

const char* what() const noexcept override;

8     Returns: An NTBS incorporating the arguments supplied in the constructor.

[Note: 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</td>
<td>Utility components</td>
</tr>
<tr>
<td>20.3</td>
<td>Compile-time integer sequences</td>
</tr>
<tr>
<td>20.4</td>
<td>Pairs</td>
</tr>
<tr>
<td>20.5</td>
<td>Tuples</td>
</tr>
<tr>
<td>20.6</td>
<td>Optional objects</td>
</tr>
<tr>
<td>20.7</td>
<td>Variants</td>
</tr>
<tr>
<td>20.8</td>
<td>Storage for any type</td>
</tr>
<tr>
<td>20.9</td>
<td>Fixed-size sequences of bits</td>
</tr>
<tr>
<td>20.10</td>
<td>Memory</td>
</tr>
<tr>
<td>20.11</td>
<td>Smart pointers</td>
</tr>
<tr>
<td>20.12</td>
<td>Memory resources</td>
</tr>
<tr>
<td>20.13</td>
<td>Scoped allocators</td>
</tr>
<tr>
<td>20.14</td>
<td>Function objects</td>
</tr>
<tr>
<td>20.15</td>
<td>Type traits</td>
</tr>
<tr>
<td>20.16</td>
<td>Compile-time rational arithmetic</td>
</tr>
<tr>
<td>20.17</td>
<td>Type indexes</td>
</tr>
<tr>
<td>20.18</td>
<td>Execution policies</td>
</tr>
<tr>
<td>20.19</td>
<td>Primitive numeric conversions</td>
</tr>
<tr>
<td>20.20</td>
<td>Formatting</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 17.10.1

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 T&& move(T&&) noexcept;
```
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 T as_const(T t) noexcept;
template<class T>
void as_const(T& 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;

template<class T, class...>
struct integer_sequence;
template<size_t... I>
using index_sequence = integer_sequence<size_t, I...>;
template<class T, size_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>;

// 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>;

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;

// 20.4.5, pair piecewise construction

struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
}:
inline constexpr piecewise_construct_t piecewise_construct{};

// defined in <tuple> (20.5.2)

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;
}:
template<class T> 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;
}:
template<
size_t I>
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.

§ 20.2.2
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 (16.5.4.2.1). The expression inside noexcept is equivalent to:

\[
\text{is_nothrow_move_constructible}_v<T> \&\& \text{is_nothrow_move_assignable}_v<T>
\]

\[
\text{template<class T, size_t N>}
\text{constexpr void swap(T \&a[N], T \&b[N]) noexcept(is_nothrow_swappable}_v<T>);\]

Constraints: \text{is_swappable}_v<T> is true.

Preconditions: a[i] is swappable with (16.5.3.2) b[i] for all i in the range \([0, N)\).

Effects: As if by swap_ranges(a, a + N, b).

20.2.3 exchange

\[
\text{template<class T, class U = T>}
\text{constexpr T exchange(T& obj, U&& new_val);}
\]

Effects: Equivalent to:

\[
\text{T old_val} = \text{std::move(obj)};
\text{obj} = \text{std::forward<U>(new_val)};
\text{return old_val;}
\]

20.2.4 Forward/move helpers

\[
\text{template<class T> constexpr T&& forward(remove_reference_t<T>& t) noexcept;}
\text{template<class T> constexpr T&& forward(remove_reference_t<T>&& t) noexcept;}
\]

Mandates: For the second overload, \text{is_lvalue_reference}_v<T> is false.

Returns: \text{static_cast<T&&>(t)}.

[Example:

\[
\text{template<class T, class A1, class A2>}
\text{shared_ptr<T> factory(A1&& a1, A2&& a2) {}
\text{return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2))};
\}}
\]

\[
\text{struct A { A(int&, const double&); }}
\]

\[
\text{void g() {}
\text{shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&}
\text{int i = 2;}
\text{shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK}
\}}
\]

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]

\[
\text{template<class T> constexpr remove_reference_t<T>&& move(T&& t) noexcept;}
\]

Returns: \text{static_cast<remove_reference_t<T>&&>(t)}.

[Example:}
template<class T, class A1>
shared_ptr<T> factory(A1&& a1) {
    return shared_ptr<T>(new T(std::forward<A1>(a1)));
}

struct A {
    A();
    A(const A&);  // copies from lvalues
    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 \texttt{A&}, so a is forwarded as a non-const lvalue. This binds to the constructor \texttt{A(const A&)}, which copies the value from a. In the second call to factory, because of the call \texttt{std::move(a)}, A1 is deduced as \texttt{A}, so a is forwarded as an rvalue. This binds to the constructor \texttt{A(A&&)}, which moves the value from a. —end example]
template<class T, class U>
constexpr bool cmp_not_equal(T t, U u) noexcept;
  \textit{Effects:} Equivalent to: \texttt{return !cmp_equal(t, u);};

template<class T, class U>
constexpr bool cmp_less(T t, U u) noexcept;
  \textit{Mandates:} Both \(T\) and \(U\) are standard integer types or extended integer types (6.8.1).
  \textit{Effects:} Equivalent to:
  \begin{verbatim}
  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);
  \end{verbatim}

template<class T, class U>
constexpr bool cmp_greater(T t, U u) noexcept;
  \textit{Effects:} Equivalent to: \texttt{return cmp_less(u, t);};

template<class T, class U>
constexpr bool cmp_less_equal(T t, U u) noexcept;
  \textit{Effects:} Equivalent to: \texttt{return !cmp_greater(t, u);};

template<class T, class U>
constexpr bool cmp_greater_equal(T t, U u) noexcept;
  \textit{Effects:} Equivalent to: \texttt{return !cmp_less(t, u);};

template<class R, class T>
constexpr bool in_range(T t) noexcept;
  \textit{Mandates:} Both \(T\) and \(R\) are standard integer types or extended integer types (6.8.1).
  \textit{Effects:} Equivalent to:
  \begin{verbatim}
  return cmp_greater_equal(t, numeric_limits<R>::min()) &&
              cmp_less_equal(t, numeric_limits<R>::max());
  \end{verbatim}

[\textit{Note:} These function templates cannot be used to compare \texttt{byte}, \texttt{char}, \texttt{char8_t}, \texttt{char16_t}, \texttt{char32_t}, \texttt{wchar_t}, and \texttt{bool}. — end note]

\section{Compile-time integer sequences}  \hfill \texttt{[intseq]}
\subsection{In general} \hfill \texttt{[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. \textit{[Note:} The \texttt{index_sequence} alias template is provided for the common case of an integer sequence of type \texttt{size_t}; see also 20.5.5. — end note]\textit{]

\subsection{Class template \texttt{integer_sequence}} \hfill \texttt{[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); }
  };
}

\textit{Mandates:} \(T\) is an integer type.
20.3.3 Alias template make_integer_sequence

```
template<class T, T N>
using make_integer_sequence = integer_sequence<T, see below>;
```

1. **Mandates:** \( N \geq 0 \).

2. The alias template `make_integer_sequence` denotes a specialization of `integer_sequence` with \( N \) non-type template arguments. The type `make_integer_sequence<T, N>` is an alias for the type `integer_sequence<T, 0, 1, \ldots, N-1>`. [Note: `make_integer_sequence<int, 0>` is an alias for the type `integer_sequence<int>`. — end note]

20.4 Pairs

20.4.1 In general

1. 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());
        constexpr explicit(const T1& x, const T2& y);
        template<class U1, class U2>
        constexpr implicit(U1&& x, U2&& y);
        template<class U1, class U2>
        constexpr implicit(const pair<U1, U2>& p);
        template<class U1, class U2>
        constexpr implicit(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: 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 first with std::forward<U1>(x) and second with std::forward<U2>(y).

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,

§ 20.4.2 568
tuple<Args1...> first_args, tuple<Args2...> second_args);

Mandates:
(20.1) is_constructible_v<first_type, Args1...> is true and
(20.2) is_constructible_v<second_type, Args2...> is true.

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.

constexpr pair& operator=(const pair& p);

Effects: Assigns p.first to first and p.second to second.

Remarks: This operator is defined as deleted unless is_copyAssignable_v<first_type> is true and is_copyAssignable_v<second_type> is true.

Returns: *this.

template<class U1, class U2> constexpr pair& operator=(const pair<U1, U2>& p);

Constraints:
(25.1) is_assignable_v<first_type&, const U1&> is true and
(25.2) is_assignable_v<second_type&, const U2&> is true.

Effects: Assigns p.first to first and p.second to second.

Returns: *this.

constexpr pair& operator=(pair&& p) noexcept(see below);

Constraints:
(32.1) is_assignable_v<first_type&, U1> is true and
(32.2) is_assignable_v<second_type&, U2> is true.

Effects: Assigns to first with std::forward<first_type>(p.first) and to second with std::forward<second_type>(p.second).

Returns: *this.

Remarks: The expression inside noexcept is equivalent to:
    is_nothrow_moveAssignable_v<T1> && is_nothrow_moveAssignable_v<T2>

template<class U1, class U2> constexpr pair& operator=(pair<U1, U2>&& p);

Constraints:
(32.1) is_assignable_v<first_type&, U1> is true and
(32.2) is_assignable_v<second_type&, U2> is true.

Effects: Assigns to first with std::forward<U1>(p.first) and to second with std::forward<U2>(p.second).

Returns: *this.

constexpr void swap(pair& p) noexcept(see below);

Preconditions: first is swappable with (16.5.3.2) 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:
    is_nothrow_swappable_v<first_type> && is_nothrow_swappable_v<second_type>

20.4.3 Specialized algorithms [pairs.spec]

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.

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: 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]

20.4.4 Tuple-like access to pair

template<class T1, class T2>
struct tuple_size<pair<T1, T2>> : integral_constant<size_t, 2> { };
template<class T1, class T2>
constexpr T1&& get(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

struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
};

inline constexpr piecewise_construct_t piecewise_construct{};

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

20.5.1 In general

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

#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<TTypes...> 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);

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

// 20.5.7, element access
template<size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>> get(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();
  constexpr explicit(see below) tuple(const Types&...);  // only if sizeof...(Types) >= 1
  template<class... UTypes>
  constexpr explicit(see below) tuple(UTypes&&...);  // only if sizeof...(Types) >= 1
  tuple(const tuple&) = default;
  tuple(tuple&&) = default;
  template<class... UTypes>
  constexpr explicit(see below) tuple(const tuple<UTypes...>&);
  template<class... UTypes>
  constexpr explicit(see below) tuple(tuple<UTypes...>&&);
  template<class U1, class U2>
  constexpr explicit(see below) tuple(const pair<U1, U2>&);  // only if sizeof...(Types) == 2
  template<class U1, class U2>
  constexpr explicit(see below) tuple(pair<U1, U2>&&);  // only if sizeof...(Types) == 2

  // allocator-extended constructors
  template<class Alloc>
  constexpr tuple(allocator_arg_t, const Alloc& a);
  template<class Alloc>
  constexpr tuple(allocator_arg_t, const Alloc& a, const Types&...);
  template<class Alloc, class... UTypes>
  constexpr tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
  template<class Alloc>
  constexpr tuple(allocator_arg_t, const Alloc& a, const tuple&);
  template<class Alloc>
  constexpr tuple(allocator_arg_t, const Alloc& a, tuple&&);
  template<class Alloc, class... UTypes>
  constexpr tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
  template<class Alloc, class... UTypes>
  constexpr tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
  template<class Alloc, class U1, class U2>
  constexpr tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);  // only if sizeof...(Types) == 2
  template<class Alloc, class U1, class U2>
  constexpr tuple(allocator_arg_t, const Alloc& a, 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<class... UTypes>
  constexpr tuple& operator=(const tuple<UTypes...>&);
  template<class... UTypes>
  constexpr tuple& operator=(tuple<UTypes...>&&);
  template<class U1, class U2>
  constexpr tuple& operator=(const pair<U1, U2>&);  // only if sizeof...(Types) == 2
  template<class U1, class 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<class... UTypes>
tuple(UTypes...) -> tuple<UTypes...>;
tuple(pair<T1, T2>) -> tuple<T1, T2>;
tuple(allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;
tuple(allocator_arg_t, Alloc, pair<T1, T2>) -> tuple<T1, T2>;
tuple(allocator_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>;

20.5.3.1 Construction

In the descriptions that follow, let i be in the range [0, sizeof...(Types)) in order, Ti be the ith type in Types, and Ui be the ith type in a template parameter pack named UTypes, where indexing is zero-based.

For each tuple constructor, an exception is thrown only if the construction of one of the types in Types throws an exception.

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.

If is_trivially_destructible_v<Ti> is true for all Ti, then the destructor of tuple is trivial.

constexpr explicit(see below) tuple();

Constraints: is_default_constructible_v<Ti> is true for all i.
Effects: Value-initializes each element.
Remarks: The expression inside explicit evaluates to true if and only if Ti is not copy-list-initializable from an empty list for at least one i. [Note: This behavior can be implemented with a trait that checks whether a const Ti& can be initialized with {}. — end note]

constexpr explicit(see below) tuple(const Types&...);

Constraints: sizeof...(Types) ≥ 1 and is_copy_constructible_v<Ti> is true for all i.
Effects: Initializes each element with the value of the corresponding parameter.
Remarks: The expression inside explicit is equivalent to:

!conjunction_v<is_convertible<const Types&, Types>...>

template<class... UTypes> constexpr explicit(see below) tuple(UTypes&&... u);

Constraints: sizeof...(Types) equals sizeof...(UTypes) and sizeof...(Types) ≥ 1 and is_constructible_v<Ti, Ui> is true for all i.
Effects: Initializes the elements in the tuple with the corresponding value in std::forward<UTypes>(u).
Remarks: The expression inside explicit is equivalent to:

!conjunction_v<is_convertible<UTypes, Types>...>
tuple(const tuple& u) = default;

Mandates: is_copy_constructible_v<Ti> is true for all i.
Effects: Initializes each element of *this with the corresponding element of u.
tuple(tuple&& u) = default;

Constraints: is_move_constructible_v<Ti> is true for all i.
Effects: For all i, initializes the ith element of *this with std::forward<Ti>(get<i>(u)).

template<class... UTypes> constexpr explicit(see below) tuple(const tuple<UTypes&&...>& u);

Constraints:

— sizeof...(Types) equals sizeof...(UTypes) and
— 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:

!conjunction_v<is_convertible<const UTypes&, Types>...>

```cpp
template<class... UTypes> constexpr explicit(see below) tuple(tuple<UTypes...>&& u);
```

**Constraints:**

(21.1) sizeof...(Types) equals sizeof...(UTypes), and
(21.2) is_constructible_v<T_i, U_i> is true for all i, and
(21.3) 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:

!conjunction_v<is_convertible<UTypes, Types>...>

```cpp
template<class U1, class U2> constexpr explicit(see below) tuple(const pair<U1, U2>& u);
```

**Constraints:**

(24.1) sizeof...(Types) is 2,
(24.2) is_constructible_v<T_0, const U1&> is true, and
(24.3) is_constructible_v<T_1, const U2&> is true.

**Effects:** Initializes the first element with u.first and the second element with u.second.

The expression inside explicit is equivalent to:

!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:**

(27.1) sizeof...(Types) is 2,
(27.2) is_constructible_v<T_0, U1> is true, and
(27.3) 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).

The expression inside explicit is equivalent to:

!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&&);
```
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

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, sizeof...(Types))
in order, Ti be the i-th type in Types, and Ui be the i-th type in a template parameter pack named UTypes,
where indexing is zero-based.

cconstexpr tuple& operator=(const tuple& u);

Effects: Assigns each element of u to the corresponding element of *this.

Remarks: This operator is defined as deleted unless is_copy assignable_v<Ti> is true for all i.

Returns: *this.

cconstexpr tuple& operator=(tuple&& u) noexcept(see below);

Constraints: is_move assignable_v<Ti> is true for all i.

Effects: For all i, assigns std::forward<Ti>(get<i>(u)) to get<i>(*this).

Remarks: The expression inside noexcept is equivalent to the logical AND of the following expressions:

is_nothrow move assignable_v<Ti>

where Ti is the i-th type in Types.

Returns: *this.

template<class... UTypes> constexpr tuple& operator=(const tuple<UTypes...>& u);

Constraints:

— sizeof...(Types) equals sizeof...(UTypes) and
— is_assignable_v<Ti&, const U&> 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:

— sizeof...(Types) equals sizeof...(UTypes) and
— is_assignable_v<Ti&, U&> is true for all i.

Effects: For all i, assigns std::forward<Ui>(get<i>(u)) to get<i>(*this).

Returns: *this.

template<class U1, class U2> constexpr tuple& operator=(const pair<U1, U2>& u);

Constraints:

— sizeof...(Types) is 2 and
— is_assignable_v<Ti&, const U1&> is true, and

§ 20.5.3.2
— is_assignable_v<T_i&, const U2&> is true.

Effects: Assigns u.first to the first element of *this and u.second to the second element of *this.

Returns: *this.

template<class U1, class U2> constexpr tuple& operator=(pair<U1, U2>&& u);

Constraints:
— sizeof...(Types) is 2 and
— is_assignable_v<T_0&, U1> is true, and
— is_assignable_v<T_1&, U2> is true.

Effects: Assigns std::forward<U1>(u.first) to the first element of *this and std::forward<U2>(u.second) to the second element of *this.

Returns: *this.

20.5.3.3 swap

constexpr void swap(tuple& rhs) noexcept(see below);

Preconditions: Each element in *this is swappable with (16.5.3.2) the corresponding element in rhs.

Effects: Calls swap for each element in *this and its corresponding element in rhs.

Remarks: The expression inside noexcept is equivalent to the logical and of the following expressions:

is_nothrow_swappable_v<T_i>

where T_i is the i-th type in Types.

Throws: Nothing unless one of the element-wise swap calls throws an exception.

20.5.4 Tuple creation functions

template<class... TTypes>
constexpr tuple<unwrap_ref Decay_t<TTypes>...> make_tuple(TTypes&&... t);

Returns: tuple<unwrap_ref Decay_t<TTypes>...>(std::forward<TTypes>(t) ...).

[Example:

int i; float j;
make_tuple(1, ref(i), cref(j))
creates a tuple of type tuple<int, int&, const float&>. — end example]

template<class... TTypes>
constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&... t) noexcept;

Effects: Constructs a tuple of references to the arguments in 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) ...).

template<class... TTypes>
constexpr tuple<TTypes&...> tie(TTypes&... t) noexcept;

Returns: tuple<TTypes&...>(t ...). When an argument in t is ignore, assigning any value to the corresponding tuple element has no effect.

[Example: tie functions allow one to create tuples that unpack tuples into variables. ignore can be used for elements that are not needed:

int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
— end example]
template<class... Tuples>
constexpr tuple<CTypes...> tuple_cat(Tuples&&... tpls);

In the following paragraphs, let \( T_i \) be the \( i \)th type in \( Tuples \), \( U_i \) be \textit{remove reference} \( T_i \), and \( tp_i \) be the \( i \)th parameter in the function parameter pack \( tpls \), where all indexing is zero-based.

\( \textbf{Preconditions:} \) For all \( i \), \( U_i \) is the type \textit{cv} \( T_i \), \textit{tuple<Args_0,...,Args_{n-1}>}, where \( cv_i \) is the (possibly empty) \( i \)th \textit{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 \( Arg_i \). For all \( A_{ik} \) the following requirements are met:

\[
\begin{align*}
\text{(9.1)} & \quad \text{If } T_i \text{ is deduced as an lvalue reference type, then } \text{is constructible}_v<A_{ik}, \text{cv}_{i} A_{ik}>& = \text{true, otherwise} \\
\text{(9.2)} & \quad \text{is constructible}_v<A_{ik}, \text{cv}_{i} A_{ik}&&>& = \text{true}.
\end{align*}
\]

\( \textbf{Remarks:} \) The types in \( CTuples \) are equal to the ordered sequence of the extended types \( Args_0,...,Args_{n-1} \), where \( n \) is equal to \textit{sizeof...(Tuples)}. Let \( e_i,... \) be the \( i \)th ordered sequence of tuple elements of the resulting \textit{tuple} object corresponding to the type sequence \( Arg_i \).

\( \textbf{Returns:} \) A \textit{tuple} object constructed by initializing the \( k_i \)th type element \( e_{ik} \) in \( e_i,... \) with

\[
\text{get}_{k_i}(\text{std::forward<T_i>}(tp_i))
\]
for each valid \( k_i \) and each group \( e_i \), in order.

\[\text{Note:} \] An implementation may support additional types in the template parameter pack \( Tuples \) that support the \textit{tuple}-like protocol, such as \textit{pair} and \textit{array}. —end note

\section{20.5.5 Calling a function with a \textit{tuple} of arguments} \([\text{tuple.apply}]\)

\[
\text{template<class F, class Tuple>}
\text{constexpr decltype(auto) apply(F&& f, Tuple&& t);}\]

\( \textbf{Effects:} \) Given the exposition-only function:

\[
\text{template<class F, class Tuple, size_t... I>}
\text{constexpr decltype(auto) apply-impl(F&& f, Tuple&& t, index_sequence<I...>)} \{ \\
\text{\hspace{1em} // exposition only} \\
\text{\hspace{1em} return INVOKE(\text{std::forward<F>}(f), \text{std::get<I>}(\text{std::forward<Tuple>}(t)))...); // see 20.14.3} \\
\} \\
\]

Equivalent to:

\[
\text{return apply-impl(\text{std::forward<F>}(f), \text{std::forward<Tuple>}(t),} \\
\text{\hspace{1em} make_index_sequence<tuple_size_v<\text{remove reference}_T<Tuple>>>{}});\]

\[
\text{template<class T, class Tuple>}
\text{constexpr T make_from_tuple(Tuple&& t);}\]

\( \textbf{Effects:} \) Given the exposition-only function:

\[
\text{template<class T, class Tuple, size_t... I>}
\text{constexpr T make-from-tuple-impl(Tuple&& t, index_sequence<I...>)} \{ \\
\text{\hspace{1em} // exposition only} \\
\text{\hspace{1em} return T(get_{I}(\text{std::forward<Tuple>}(t))...);} \\
\} \\
\]

Equivalent to:

\[
\text{return make-from-tuple-impl<T>(} \\
\text{\hspace{1em} forward<Tuple>(t),} \\
\text{\hspace{1em} make_index_sequence<tuple_size_v<\text{remove reference}_T<Tuple>>>{}}); \\
\]

\[\text{Note:} \] The type of \( T \) must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. —end note

\section{20.5.6 Tuple helper classes} \([\text{tuple.helper}]\)

\[
\text{template<class T> struct tuple_size;}\]

All specializations of \textit{tuple size} meet the \textit{Cpp17UnaryTrait} requirements (20.15.1) with a base characteristic of \textit{integral_constant<\textit{size_t}, N>} for some \( N \).
template<class... Types>
    struct tuple_size<tuple<Types...>> : public integral_constant<size_t, sizeof...(Types)> { };
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]
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: 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.

Effects: 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: The above definition does not require t.tail (or u.tail) to be constructed. It may not even be possible, as t and u are not required to be copy constructible. Also, all comparison 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: 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.

Remarks: The expression inside noexcept is equivalent to:

noexcept((x.swap(y))

Effects: As if by x.swap(y).
20.6 Optional objects [optional]

20.6.1 In general [optional.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 [optional.syn]

```cpp
#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;
    inline constexpr nullopt_t nullopt;

    // 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, class 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, class U>
    constexpr compare_three_way_result_t<T,U>
    operator<=>(const optional<T>&, const U&);
}
```

§ 20.6.2
// 20.6.9, specialized algorithms
template<class T>
void swap(optional<T>&, optional<T>&) noexcept(see below);

template<class T>
constexpr optional<T>& make_optional(T&&);

template<class T, class... Args>
explicit optional(T&&...);

template<class T, class U, class... Args>
explicit optional(initializer_list<U>, Args&&...);

// 20.6.10, hash support
template<class T>
struct hash;

20.6.3 Class template optional

namespace std {

template<class T>
class optional {
public:

    using value_type = T;

    // 20.6.3.1, constructors
    constexpr optional() noexcept;
    constexpr optional(nullopt_t) noexcept;
    constexpr optional(const optional&);
    constexpr optional(optional&&) noexcept(see below);
    template<class... Args>
    explicit optional(in_place_t, Args&&...);
    template<class U, class... Args>
    explicit optional(in_place_t, initializer_list<U>, Args&&...);
    template<class U = T>
    explicit(see below) optional(U&&);
    template<class U>
    explicit(see below) optional(optional<U>&);

    // 20.6.3.2, destructor
    ~optional();

    // 20.6.3.3, assignment
    optional& operator=(nullopt_t) noexcept;
    constexpr optional& operator=(optional&) noexcept;
    constexpr optional& operator=(optional&&) noexcept(see below);
    template<class U = T> optional& operator=(U&&);
    template<class U> optional& operator=(optional<U>&);
    template<class U = T> optional& operator=(const optional<U>&);
    template<class U> optional& operator=(optional<U>&&);
    template<class... Args> T& emplace(Args&&...);
    template<class U, class... Args> T& emplace(initializer_list<U>, Args&&...);

    // 20.6.3.4, swap
    void swap(optional&) noexcept(see below);

    // 20.6.3.5, observers
    constexpr const T* operator->() const;
    constexpr T* operator->() const;
    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&&;

template<class U>
constexpr T value_or(U&&) const&;

template<class U>
constexpr T value_or(U&&) &&;

} // 20.6.3.6, modifiers

void reset() noexcept;

private:
    T *val; // exposition only
};

template<class T>
optional(T) -> optional<T>;

1 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`.

2 Member `val` is provided for exposition only. When an `optional<T>` object contains a value, `val` points to the contained value.

3 `T` shall be a type other than `cv in_place_t` or `cv nullopt_t` that meets the `Cpp17Destructible` requirements (Table 32).

20.6.3.1 Constructors [optional.ctor]

constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;

Postconditions: `*this` does not contain a value.

Remarks: No contained value is initialized. For every object type `T` these constructors are constexpr constructors (9.2.5).

constexpr optional(const optional& rhs);

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: \( \text{bool}(\text{rhs}) == \text{bool}(\text{*this}) \).

Throws: Any exception thrown by the selected constructor of `T`.

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);

Constraints: `is_move_constructible_v<T>` is true.

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.

Postconditions: \( \text{bool}(\text{rhs}) == \text{bool}(\text{*this}) \).

Throws: Any exception thrown by the selected constructor of `T`.

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.
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(optional<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_constructible_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>

§ 20.6.3.1 584
template<class U> explicit(see below) optional(optional<U>&& rhs);

Constraints:

- is_constructible_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, and
- is_convertible_v<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 std::move(*rhs). bool(rhs) is unchanged.

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<U, T>

20.6.3.2 Destructor [optional.dtor]

~optional();

Effects: If is_trivially_destructible_v<T> != true and *this contains a value, calls val->T::~T()

Remarks: If is_trivially_destructible_v<T> is true, then this destructor is trivial.

20.6.3.3 Assignment [optional.assign]

optional<T>& operator=(nullopt_t) noexcept;

Effects: If *this contains a value, calls val->T::~T() to destroy the contained value; otherwise no effect.

Postconditions: *this does not contain a value.

Returns: *this.

constexpr optional<T>& operator=(const optional& rhs);

Effects: See Table 42.

Table 42: optional::operator=(const optional&) effects [tab:optional.assign.copy]

<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></td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with *rhs</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>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 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;

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>*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>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>destroys the contained value by calling val-&gt;T::~T()</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.
(19.7) is_convertible_v<optional<U>&, T> is false,
(19.8) is_convertible_v<optional<U>&&, T> is false,
(19.9) is_convertible_v<const optional<U>&, T> is false,
(19.10) is_convertible_v<const optional<U>&&, T> is false,
(19.11) is_assignable_v<T&, optional<U>&> is false,
(19.12) is_assignable_v<T&, optional<U>&&> is false,
(19.13) is_assignable_v<T&, const optional<U>&> is false, and
(19.14) 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></td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with *rhs</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>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:

(24.1) is_constructible_v<T, U> is true,
(24.2) is_assignable_v<T&, U> is true,
(24.3) is_constructible_v<T, optional<U>&> is false,
(24.4) is_constructible_v<T, optional<U>&&> is false,
(24.5) is_constructible_v<T, const optional<U>&> is false,
(24.6) is_constructible_v<T, const optional<U>&&> is false,
(24.7) is_convertible_v<optional<U>&, T> is false,
(24.8) is_convertible_v<optional<U>&&, T> is false,
(24.9) is_convertible_v<const optional<U>&, T> is false,
(24.10) is_convertible_v<const optional<U>&&, T> is false,
(24.11) is_assignable_v<T&, optional<U>&> is false,
(24.12) is_assignable_v<T&, optional<U>&&> is false,
(24.13) is_assignable_v<T&, const optional<U>&> is false, and
(24.14) 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>*<em>assigns std::move(<em>rhs) to the contained value</em></em></td>
</tr>
<tr>
<td><strong>rhs does not contain a value</strong></td>
<td><strong>destroys the contained value by calling val-&gt;T::~T()</strong></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.

```cpp
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.

```cpp
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.

20.6.3.4 Swap

```cpp
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>*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><strong>rhs does not contain a value</strong></td>
</tr>
<tr>
<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>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>

### 20.6.3.5 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 T&& operator*() &&;
constexpr const T&& operator*() const&&;
```

5. **Preconditions:** *this contains a value.
6. **Effects:** Equivalent to: return `std::move(*val)`;
7. **Remarks:** These functions are constexpr functions.

```cpp
constexpr explicit operator bool() const noexcept;
```

9. **Returns:** true if and only if *this contains a value.
10. **Remarks:** This function is a constexpr function.

```cpp
constexpr bool has_value() const noexcept;
```

11. **Returns:** true if and only if *this contains a value.
12. **Remarks:** This function is a constexpr function.

```cpp
constexpr const T& value() const&;
```

13. **Remarks:** This function is a constexpr function.
constexpr T& value() &;

Effects: Equivalent to:

return bool(*this) ? *val : throw bad_optional_access();

constexpr T&& value() &&;
constexpr const T&& value() const&&;

Effects: Equivalent to:

return bool(*this) ? std::move(*val) : throw bad_optional_access();

template<class U> constexpr T value_or(U&& v) const&;

Mandates: is_copy_constructible_v<T> && is_convertible_v<U&&, T> is true.

Effects: Equivalent to:

return bool(*this) ? **this : static_cast<T>(std::forward<U>(v));

template<class U> constexpr T value_or(U&& v) &&;

Mandates: is_move_constructible_v<T> && is_convertible_v<U&&, T> is true.

Effects: Equivalent to:

return bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v));

20.6.3.6 Modifiers [optional.mod]

void reset() noexcept;

Effects: If *this contains a value, calls val->T::T() to destroy the contained value; otherwise no effect.

Postconditions: *this does not contain a value.

20.6.4 No-value state indicator [optional.nullopt]

struct nullopt_t{see below};
inline constexpr nullopt_t nullopt(unspecified);

The struct nullopt_t is an empty class type used as a unique type to indicate the state of not containing a value for optional objects. In particular, optional<T> has a constructor with nullopt_t as a single argument; this indicates that an optional object not containing a value shall be constructed.

Type nullopt_t shall not have a default constructor or an initializer-list constructor, and shall not be an aggregate.

20.6.5 Class bad_optional_access [optional.bad.access]

class bad_optional_access : public exception {
public:
    // see 17.9.2 for the specification of the special member functions
    const char* what() const noexcept override;
};

The class 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.

const char* what() const noexcept override;

Returns: An implementation-defined NTBS.

20.6.6 Relational operators [optional.relops]
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. [Note: T need not be Cpp17EqualityComparable. — end note]

Returns: If bool(x) != bool(y), false; otherwise if bool(x) == false, 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 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, three_way_comparable_with<T> 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 == *y is well-formed and its result is convertible to bool. [Note: 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& 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;`

```cpp
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;`

```cpp
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;`

```cpp
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;`

```cpp
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;`

```cpp
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;`

```cpp
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;`

```cpp
template<class T, class 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;`

20.6.9 Specialized algorithms
[optional.specalg]

```cpp
template<class T> void swap(optional<T>& x, optional<T>& y) noexcept(noexcept(x.swap(y)));
```

Constraints: `is_move_constructible_v<T>` is true and `is_swappable_v<T>` is true.
Effects: Calls `x.swap(y)`.

```cpp
template<class T> constexpr optional<decay_t<T>> make_optional(T&& v);
```

Returns: `optional<decay_t<T>>(std::forward<T>(v))`.

§ 20.6.9
template<class T, class... Args>
constexpr optional<T> make_optional(Args&&... args);

4 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);

5 Effects: Equivalent to: return optional<T>(in_place, il, std::forward<Args>(args)...);

20.6.10 Hash support [optional.hash]

template<class T> struct hash<optional<T>>;

1 The specialization hash<optional<T>> is enabled (20.14.18) 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 [variant]

20.7.1 In general [variant.general]

1 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 [variant.syn]

#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;
    template<class T> struct variant_size<variant 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;
    template<size_t I, class... Types>
    using variant_alternative_t = typename variant_alternative<I, Types...>::type;

    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... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&);

    template<size_t I, class... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&&);

    template<size_t I, class... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&&);

    template<size_t I, class... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&&&);

    template<size_t I, class... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&);

    template<size_t I, class... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&&);

    template<size_t I, class... Types>
    constexpr bool holds_alternative(const variant_alternative_t<I, variant<Types...>>&&&);
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;

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...>&);

template<class... Types> requires (three_way_comparable<Types> && ...)
  constexpr common_comparison_category_t<compare_three_way_result_t<Types>...>
  operator<=>(const variant<Types...>&, const variant<Types...>&);

// 20.7.7, visitation
template<class Visitor, class... Variants>
  constexpr
      see below visit(Visitor&&, Variants&&...);

template<class R, class Visitor, class... Variants>
  constexpr R 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
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...>>;
20.7.3 Class template variant

namespace std {
    template<class... Types>
    class variant {
        public:
            // 20.7.3.1, 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&&...);
            // 20.7.3.2, destructor
            ~variant();
            // 20.7.3.3, assignment
            constexpr variant& operator=(const variant&);
            constexpr variant& operator=(variant&&) noexcept;
            template<class T> variant& operator=(T&&) noexcept;
            // 20.7.3.4, 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&&...);
            // 20.7.3.5, value status
            constexpr bool valueless_by_exception() const noexcept;
            constexpr size_t index() const noexcept;
            // 20.7.3.6, 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).

3 A program that instantiates the definition of `variant` with no template arguments is ill-formed.
20.7.3.1 Constructors

In the descriptions that follow, let \( i \) be in the range \([0, \text{sizeof}...\text{Types})\), and \( T_i \) be the \( i^{\text{th}} \) type in Types.

```
constexpr variant() noexcept(see below);
```

- **Constraints**: \( \text{is\_default\_constructible\_v<T_0>} \) is true.
- **Effects**: Constructs a variant holding a value-initialized value of type \( T_0 \).
- **Postconditions**: \( \text{valueless\_by\_exception()} \) is false and \( \text{index()} \) 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 \( \text{is\_nothrow\_default\_constructible\_v<T_0>} \). [Note: See also class monostate. — end note]

```
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 \( \text{get}<j>(w) \), where \( j \) is \( w.\text{index()} \). 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 \( \text{is\_copy\_constructible\_v<T_i>} \) is true for all \( i \). If \( \text{is\_trivially\_copy\_constructible\_v<T_i>} \) is true for all \( i \), this constructor is trivial.

```
constexpr variant(variant&& w) noexcept(see below);
```

- **Constraints**: \( \text{is\_move\_constructible\_v<T_i>} \) 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 \( \text{get}<j>(\text{std::move}(w)) \), where \( j \) is \( w.\text{index()} \). Otherwise, initializes the variant to not hold a value.
- **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 \( \text{is\_nothrow\_move\_constructible\_v<T_i>} \) for all \( i \). If \( \text{is\_trivially\_move\_constructible\_v<T_i>} \) is true for all \( i \), this constructor is trivial.

```
template<class T> constexpr variant(T&& t) noexcept(see below);
```

- **Constraints**: \( \text{- (15.1)} \) sizeof...(Types) is nonzero,
  \( \text{- (15.2)} \) \( \text{is\_same\_v<\text{remove\_cvref\_t<T_i>, variant}>} \) is false,
  \( \text{- (15.3)} \) \( \text{remove\_cvref\_t<T_i>} \) is neither a specialization of \( \text{in\_place\_type\_t} \) nor a specialization of \( \text{in\_place\_index\_t} \),
  \( \text{- (15.4)} \) \( \text{is\_constructible\_v<T_j>, T_i>} \) is true, and
  \( \text{- (15.5)} \) the expression \( \text{FUN}(\text{std::forward<T_i>(t)}) \) (with \( \text{FUN} \) being the above-mentioned set of imaginary functions) is well-formed. [Note: \( \text{variant<\text{string, string}> v("abc")}; \)
  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 \( \text{std::forward<T_i>(t)} \).
- **Postconditions**: \( \text{holds\_alternative<T_j>(\*this)} \) 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... and
(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... and
(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_I’s selected constructor is a constexpr constructor, this constructor is a constexpr constructor.
20.7.3.2 Destructor

*/variant();

1 Effects: If valueless_by_exception() is false, destroys the currently contained value.
2 Remarks: If is_trivially_destructible_v<T_i> is true for all T_i, then this destructor is trivial.

20.7.3.3 Assignment

constexpr variant& operator=(const variant& rhs);

1 Let j be rhs.index().
2 Effects:

(2.1) — If neither *this nor rhs holds a value, there is no effect.
(2.2) — Otherwise, if *this holds a value but rhs does not, destroys the value contained in *this and
sets *this to not hold a value.
(2.3) — Otherwise, if index() == j, assigns the value contained in rhs to the value contained in *this.
(2.4) — Otherwise, if either is_nothrow_copy_constructible_v<T_j> is true or is_nothrow_move_constructible_v<T_j> is false, equivalent to emplace<j>(get<j>(rhs)).
(2.5) — Otherwise, equivalent to operator=(variant(rhs)).
3 Returns: *this.
4 Postconditions: index() == rhs.index().
5 Remarks: This operator is defined as deleted unless is_copy_constructible_v<T_i> && is_copy_assignable_v<T_i> is true for all i. If is_trivially_copy_constructible_v<T_i> && is_trivially_copy_assignable_v<T_i> && is_trivially_destructible_v<T_i> is true for all i, this assignment operator is trivial.

constexpr variant& operator=(variant&& rhs) noexcept(see below);

1 Let j be rhs.index().
2 Constraints: is_move_constructible_v<T_j> && is_move_assignable_v<T_j> is true for all i.
3 Effects:

(8.1) — If neither *this nor rhs holds a value, there is no effect.
(8.2) — Otherwise, if *this holds a value but rhs does not, destroys the value contained in *this and
sets *this to not hold a value.
(8.3) — Otherwise, if index() == j, assigns get<j>(std::move(rhs)) to the value contained in *this.
(8.4) — Otherwise, equivalent to emplace<j>(get<j>(std::move(rhs))).
4 Returns: *this.
5 Remarks: If is_trivially_move_constructible_v<T_j> && is_trivially_move_assignable_v<T_j> && is_trivially_destructible_v<T_j> is true for all i, this assignment operator is trivial. The expression inside noexcept is equivalent to: is_nothrow_move_constructible_v<T_j> && is_nothrow_move_assignable_v<T_j> for all i.

(10.1) — If an exception is thrown during the call to T_j’s move construction (with j being rhs.index()),
the 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; index() will be j.

template<class T> variant& operator=(T&& t) noexcept(see below);

1 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 x[] = {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 assignment.
2 Constraints:

(12.1) — is_same_v<remove_cvref_t<T>, variant> is false,
— is_assignable_v<Tj, T> && is_constructible_v<Tj, T> is true, and
— the expression FUN(std::forward<T>(t)) (with FUN being the above-mentioned set of imaginary functions) is well-formed. [Note:
  variant<string, string> v;
  v = "abc";
  is ill-formed, as both alternative types have an equally viable constructor for the argument. — end note]

Effects:
(13.1) — If *this holds a Tj, assigns std::forward<T>(t) to the value contained in *this.
(13.2) — Otherwise, if is_nothrow_constructible_v<Tj, T> || !is_nothrow_move_constructible_v<Tj, T> is true, equivalent to emplace<std::forward<T>(t)).
(13.3) — Otherwise, equivalent to operator=(variant(std::forward<T>(t))).

Postconditions: holds_alternative<Tj>(*this) is true, with Tj selected by the imaginary function overload resolution described above.

Returns: *this.

Remarks: The expression inside noexcept is equivalent to:
  is_nothrow_assignable_v<Tj, T> && is_nothrow_constructible_v<Tj, T>
(16.1) — If an exception is thrown during the assignment of std::forward<T>(t) to the value contained in *this, the state of the contained value and t are as defined by the exception safety guarantee of the assignment expression; valueless_by_exception() will be false.
(16.2) — If an exception is thrown during the initialization of the contained value, the variant object might not hold a value.

§ 20.7.3.4 Modifiers

template<class T, class... Args> T& emplace(Args&&... args);

Constraints: is_constructible_v<T, Args...> is true, and T occurs exactly once in Types.

Effects: Equivalent to:
  return emplace<I>(std::forward<Args>(args)...);
where I is the zero-based index of T in Types.

template<class T, class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);

Constraints: is_constructible_v<T, initializer_list<U>&, Args...> is true, and T occurs exactly once in Types.

Effects: Equivalent to:
  return emplace<I>(il, std::forward<Args>(args)...);
where I is the zero-based index of T in Types.

template<
  size_t I, class... Args>
  variant_alternative_t<I, variant<Types...>>& emplace(Args&&... args);

Mandates: I < sizeof...(Types).

Constraints: is_constructible_v<TI, 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 TI with the arguments 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 might not hold a value.
template<size_t I, class U, class... Args>
    variant_alternative_t<I, variant<Types...>>& emplace(initializer_list<U> il, Args&&... args);

Mandates: I < sizeof...(Types).

Constraints: is_constructible_v<T_i, initializer_list<U>> && 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 il, std::forward<Args>(args)....

Postconditions: index() is I.

Returns: A reference to the new contained value.

Remarks: If an exception is thrown during the initialization of the contained value, the variant might not hold a value.

20.7.3.5 Value status

constexpr bool valueless_by_exception() const noexcept;

Effects: Returns false if and only if the variant holds a value.

[Note: 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]

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.

20.7.3.6 Swap

void swap(variant& rhs) noexcept(see below);

Mandates: is_move_constructible_v<T_i> is true for all i.

Preconditions: Lvalues of type T_i are swappable (16.5.3.2).

Effects:

(3.1) If valueless_by_exception() && rhs.valueless_by_exception() no effect.

(3.2) Otherwise, if index() == rhs.index(), calls swap(get<i>(*this), get<i>(rhs)) where i is index().

(3.3) 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<T_i> && is_nothrow_swappable_v<T_i> for all i.

20.7.4 variant helper classes

template<class T>
    struct variant_size;

All specializations of variant_size meet the Cpp17UnaryTypeTrait requirements (20.15.1) with a base characteristic of integral_constant<size_t, N> for some N.
template<class T> class 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 (20.15.1) with a base characteristic of \texttt{integral_-constant}<\texttt{size_t}, VS::value>.

\texttt{template<class... Types>}
\begin{verbatim}
struct variant_size<variant<Types...>> : integral_constant<\texttt{size_t}, sizeof...(Types)> { };
\end{verbatim}

\texttt{template<\texttt{size_t} I, class T> class 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 (20.15.1) with a member typedef \texttt{type} that names the type \texttt{add_const_t}<VA::\texttt{type}>

\begin{verbatim}
variant_alternative<I, variant<Types...>>::\texttt{type}
\end{verbatim}

\begin{itemize}
  \item \textbf{Mandates:} I < sizeof...(Types).
  \item \textbf{Type:} The type T_I.
\end{itemize}

\begin{itemize}
  \item \textbf{Mandates:} The type T occurs exactly once in Types.
  \item \textbf{Returns:} true if index() is equal to the zero-based index of T in Types.
\end{itemize}

\texttt{template<\texttt{size_t} I, class... Types>}
\begin{verbatim}
constexpr bool holds_alternative(const variant<Types...>& v) noexcept;
\end{verbatim}

\begin{itemize}
  \item \textbf{Mandates:} I < sizeof...(Types).
  \item \textbf{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.
\end{itemize}

\texttt{template<class T, class... Types> constexpr T& get(variant<Types...>& v);}
\texttt{template<class T, class... Types> constexpr T&& get(variant<Types...>&& v);}
\texttt{template<class T, class... Types> constexpr const T& get(const variant<Types...>& v);}
\texttt{template<class T, class... Types> constexpr const T&& get(const variant<Types...>&& v);}

\begin{itemize}
  \item \textbf{Mandates:} The type T occurs exactly once in Types.
  \item \textbf{Effects:} If v holds a value of type T, returns a reference to that value. Otherwise, throws an exception of type bad_variant_access.
\end{itemize}

\texttt{template<\texttt{size_t} I, class... Types> constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>> get_if(variant<Types...>* v) noexcept;}
\texttt{template<\texttt{size_t} I, class... Types> constexpr add_pointer_t<const variant_alternative_t<I, variant<Types...>>> get_if(const variant<Types...>* v) noexcept;}

\begin{itemize}
  \item \textbf{Mandates:} I < sizeof...(Types).
  \item \textbf{Returns:} A pointer to the value stored in the variant, if v \(!= \texttt{nullptr}\) and v->index() == I. Otherwise, returns \texttt{nullptr}.
\end{itemize}

\texttt{template<class T, class... Types> constexpr add_pointer_t<\texttt{T}>& get_if(variant<Types...>* v) noexcept;}

\section*{§ 20.7.5 601}
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

[variant.relops]

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 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(), 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 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;
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

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);

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:

\[ \text{INVOK}e(\text{std::forward}<\text{Visitor}>\!(\text{vis}), \text{get}<m>(\text{std::forward}<\text{Variants}>\!(\text{vars})\ldots)) \] // see 20.14.3

for the first form and

\[ \text{INVOK}e<R>(\text{std::forward}<\text{Visitor}>\!(\text{vis}), \text{get}<m>(\text{std::forward}<\text{Variants}>\!(\text{vars})\ldots)) \] // see 20.14.3

for the second form.

**Mandates:** For each valid pack m, e(m) is a valid expression. All such expressions are of the same type and value category.

**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.

**Throws:** bad_variant_access if any variant in vars is valueless_by_exception().

**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. For n > 1, the invocation of the callable object has no complexity requirements.

20.7.8 Class monostate

struct monostate{};

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

constexpr bool operator==(monostate, monostate) noexcept { return true; }
constexpr strong_ordering operator<=>(monostate, monostate) noexcept
{ return strong_ordering::equal; }

[Note: monostate objects have only a single state; they thus always compare equal. — end note]

20.7.10 Specialized algorithms

template<class... Types>
void swap(variant<Types...>& v, variant<Types...>& w) noexcept(see below);

Constraints: is_move_constructible_v<T_i> && is_swappable_v<T_i> is true for all i.

Effects: Equivalent to v.swap(w).

Remarks: The expression inside noexcept is equivalent to noexcept(v.swap(w)).

20.7.11 Class bad_variant_access

class bad_variant_access : public exception {
public:
    // see 17.9.2 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.

const char* what() const noexcept override;

Returns: An implementation-defined NTBS.
20.7.12 Hash support

The specialization `hash<variant<Types...>>` is enabled (20.14.18) if and only if every specialization in `hash<remove_const_t<Types>>...` is enabled. The member functions are not guaranteed to be noexcept.

The specialization is enabled (20.14.18).

20.8 Storage for any type

This subclause describes components that C++ programs may use to perform operations on objects of a discriminated type.

[Note: The discriminated type may 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]

20.8.1 Header `<any>` synopsis

namespace std {
    // 20.8.2, class bad_any_cast
    class bad_any_cast;
    // 20.8.3, class any
    class any;
    // 20.8.4, 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;
}

20.8.2 Class bad_any_cast

Objects of type `bad_any_cast` are thrown by a failed `any_cast` (20.8.4).

const char* what() const noexcept override;

Returns: An implementation-defined NTBS.
20.8.3 Class any

namespace std {
    class any {
        public:
            // 20.8.3.1, 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();
            // 20.8.3.2, assignments
            any& operator=(const any& rhs);
            any& operator=(any&& rhs) noexcept;
            template<class T>
            any& operator=(T&& rhs);
            // 20.8.3.3, 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;
            // 20.8.3.4, observers
            bool has_value() const noexcept;
            const type_info& type() const noexcept;
    }
}

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.

The non-member any_cast functions provide type-safe access to the contained value.

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: A contained value of type int could be stored in an internal buffer, not in separately-allocated memory. — end example]

20.8.3.1 Construction and destruction

constexpr any() noexcept;
1 Postconditions: has_value() is false.

any(const any& other);
2 Effects: If other.has_value() is false, constructs an object that has no value. Otherwise, equivalent to any(in_place_type_t<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().

20.8.3.2 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).

Returns: *this.

Postconditions: The state of *this is equivalent to the original state of rhs.
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.

20.8.3.3 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.

20.8.3.4 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: Useful for querying against types known either at compile time or only at runtime. — end note]

20.8.4 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:

any x(5);  // x holds int
assert(any_cast<int>(x) == 5);  // cast to value
any_cast<int&>(x) = 10;  // cast to reference
assert(any_cast<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;
template<class T>
T* any_cast(any* operand) noexcept;

 Returns: If operand != nullptr &amp; operand-&gt;type() == typeid(T), a pointer to the object contained by operand; otherwise, nullptr.

[Example:
  bool is_string(const any&amp; operand) {
    return any_cast&lt;string&gt;(&amp;operand) != nullptr;
  }
  — end example]

20.9 Bitsets

20.9.1 Header &lt;bitset&gt; synopsis

The header &lt;bitset&gt; defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```cpp
#include &lt;string&gt;
#include &lt;iostream&gt; // for istream (29.7.1), ostream (29.7.2), see 29.3.1

namespace std {
  template&lt;size_t N&gt; class bitset;
}

// 20.9.4, bitset operators
template&lt;size_t N&gt;
bitset&lt;N&gt; operator&(const bitset&lt;N&gt;&, const bitset&lt;N&gt;&) noexcept;
template&lt;size_t N&gt;
bitset&lt;N&gt; operator|(const bitset&lt;N&gt;&, const bitset&lt;N&gt;&) noexcept;
template&lt;size_t N&gt;
bitset&lt;N&gt; operator^(const bitset&lt;N&gt;&, const bitset&lt;N&gt;&) noexcept;
template&lt;class charT, class traits, size_t N&gt;
basic_istream&lt;charT, traits&gt;&amp;
  operator&gt;&gt;(basic_istream&lt;charT, traits&gt;&amp; is, bitset&lt;N&gt;&amp; x);
template&lt;class charT, class traits, size_t N&gt;
basic_ostream&lt;charT, traits&gt;&amp;
  operator&gt;&lt;(basic_ostream&lt;charT, traits&gt;&amp; os, const bitset&lt;N&gt;&amp; x);
}

20.9.2 Class template bitset

namespace std {
  template&lt;size_t N&gt; class bitset {
    public:
      // bit reference
      class reference {
        friend class bitset;
        reference() noexcept;

        public:
          reference(const reference&amp;) = default;
          ~reference();
          reference&amp; operator=(bool x) noexcept; // for b[i] = x;
          reference&amp; operator=(const reference&amp;) noexcept; // for b[i] = b[j];
          bool operator~() const noexcept; // flips the bit
          operator bool() const noexcept; // for x = b[i];
          reference&amp; flip() noexcept; // for b[i].flip();
      };

      // 20.9.2.1, 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'));

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'));

// 20.9.2.2, 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
template<class T> struct hash;
template<size_t N> struct hash<bitset<N>>;

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.
The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

(3.1) an invalid-argument error is associated with exceptions of type invalid_argument (19.2.4);
(3.2) an out-of-range error is associated with exceptions of type out_of_range (19.2.6);
(3.3) an overflow error is associated with exceptions of type overflow_error (19.2.9).

### 20.9.2.1 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 `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 `str.size()` - `pos`. Initializes the first M bit positions to values determined from the corresponding characters in the string `str`. M is the smaller of N and rlen.

An element of the constructed object has value zero if the corresponding character in `str`, beginning at position `pos`, is `zero`. Otherwise, the element has the value one. Character position `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 `traits::eq` to compare the character values.

**Throws:** out_of_range if `pos > str.size()` or invalid_argument if any of the rlen characters in `str` beginning at position `pos` is other than `zero` or `one`.

```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)
```

### 20.9.2.2 Members

```cpp
bitset<N>& operator&=(const bitset<N>& rhs) noexcept;
```

**Effects:** Clears each bit in `*this` for which the corresponding bit in `rhs` is clear, and leaves all other bits unchanged.

**Returns:** `*this`.

```cpp
bitset<N>& operator|=(const bitset<N>& rhs) noexcept;
```

**Effects:** Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

§ 20.9.2.2
Returns: *this.

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.

bitset<N>& operator<(size_t pos) noexcept;

Effects: Replaces each bit at position I in *this with a value determined as follows:

(7.1) If I < pos, the new value is zero;
(7.2) If I >= pos, the new value is the previous value of the bit at position I - pos.

Returns: *this.

bitset<N>& operator>>(size_t pos) noexcept;

Effects: Replaces each bit at position I in *this with a value determined as follows:

(9.1) If pos >= N - I, the new value is zero;
(9.2) If pos < N - I, the new value is the previous value of the bit at position I + pos.

Returns: *this.

bitset<N>& set() noexcept;

Effects: Sets all bits in *this.

Returns: *this.

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.

bitset<N>& reset() noexcept;

Effects: Resets all bits in *this.

Returns: *this.

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.

bitset<N> operator~() const noexcept;

Effects: Constructs an object x of class bitset<N> and initializes it with *this.

Returns: x.flip().

bitset<N>& flip() noexcept;

Effects: Toggles all bits in *this.

Returns: *this.

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

The specialization is enabled (20.14.18).

20.9.4 bitset operators

bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) &= rhs.

bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) |= rhs.

bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) ^= rhs.

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 (29.7.4.2).

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:

(5.1) N characters have been extracted and stored;
(5.2) end-of-file occurs on the input sequence;
(5.3) 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.

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 29.7.5.2).

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
  template<class T, class Alloc>
  inline constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;

  // 20.10.8.2, 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>
  constexpr auto uses_allocator_construction_args(const Alloc& alloc) noexcept -> see below;

  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;

  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,
                                           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
// 25.11.1, special memory concepts
template<class I>
 concept no-throw-input-iterator = see below; // exposition only

template<class I>
 concept no-throw-forward-iterator = see below; // exposition only

template<class S, class I>
 concept no-throw-sentinel = see below; // exposition only

template<class R>
 concept no-throw-input-range = see below; // exposition only

template<class R>
 concept no-throw-forward-range = see below; // exposition only

namespace ranges {

template<no-throw-forward-iterator I, no-throw-sentinel<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);

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);
}

template<class NoThrowForwardIterator>
 void uninitialized_value_construct(NoThrowForwardIterator first,
 NoThrowForwardIterator last);

§ 20.10.2
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<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 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<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(IR&& 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<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);
}

namespace ranges {
    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,
    InputIterator first, InputIterator last,
    NoThrowForwardIterator result);

    template<class InputIterator, class Size, class NoThrowForwardIterator>
    NoThrowForwardIterator uninitialized_move(InputIterator first, Size n,
    NoThrowForwardIterator result);

    template<class ExecutionPolicy, class InputIterator, class Size, class NoThrowForwardIterator>
    NoThrowForwardIterator uninitialized_move(ExecutionPolicy&& exec,
    InputIterator first, Size n,
    NoThrowForwardIterator result);
}

§ 20.10.2 617
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<O> S2>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
  uninitialized_move_result<I, O>
    uninitialized_move(I first, S1 last, 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<O> S>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
  uninitialized_move_n_result<I, O>
    uninitialized_move_n(I first, iter_difference_t<I> n, O ofirst, S olast);
}

namespace ranges {
  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);

  template<class NoThrowForwardIterator, class Size, class T>
  NoThrowForwardIterator
  uninitialized_fill_n(NoThrowForwardIterator first, Size n, const T& x);

  template<class ExecutionPolicy, class NoThrowForwardIterator, class Size, class T>
  NoThrowForwardIterator
  uninitialized_fill_n(ExecutionPolicy&& exec, // see 25.3.5
                       NoThrowForwardIterator first, Size n, const T& x);
}

namespace ranges {
  template<no-throw-forward-iterator I, no-throw-sentinel<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);
}
// 25.11.7, 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);
}

// 25.11.8, destroy
namespace ranges {
  template<destructible T>
  constexpr void destroy_at(T* location) noexcept;
  template<no-throw-input-iterator I, no-throw-sentinel<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;
  template<no-throw-input-iterator I>
  requires destructible<iter_value_t<I>>
  constexpr I destroy_n(I first, iter_difference_t<I> n) noexcept;
}

// 20.11.1, class template unique_ptr
template<class T> struct default_delete;
template<class T> struct default_delete<T[]>;
template<class T, class D = default_delete<T[]>> class unique_ptr;
template<class T, class D> class unique_ptr<T[], D>;

// T is not array
template<class T, class... Args>
unique_ptr<T> make_unique(Args&&... args);

// T is U[]
template<class T>
unique_ptr<T> make_unique(size_t n);

// T is U[N]
template<class T, class... Args>
unspecified make_unique(Args&&...) = delete;

// T is not array
template<class T>
unique_ptr<T> make_unique_for_overwrite();

// T is U[]
template<class T>
unique_ptr<T> make_unique_for_overwrite(size_t n);

// T is U[N]
template<class T, class... Args>
unspecified make_unique_for_overwrite(Args&&...) = delete;

template<class T, class D>
void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

// T is not array
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>
    bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

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);

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;

// 20.11.3.6, 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[N]

template<class T, class A>
    shared_ptr<T> allocate_shared(const A& a, const remove_extent_t<T>& u); // T is U[N]

template<class T>
    shared_ptr<T> make_shared_for_overwrite(); // T is not U[]

template<class T, class A>
    shared_ptr<T> allocate_shared_for_overwrite(const A& a); // T is not U[]

template<class T>
    shared_ptr<T> make_shared_for_overwrite(size_t N); // T is U[]

template<class T, class A>
    shared_ptr<T> allocate_shared_for_overwrite(const A& a, size_t N); // T is U[]

// 20.11.3.7, 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;

template<class T>
    bool operator==(const shared_ptr<T>& x, nullptr_t) noexcept;

template<class T>
    strong_ordering operator<=>(const shared_ptr<T>& x, nullptr_t) noexcept;

// 20.11.3.8, shared_ptr specialized algorithms
template<class T>
    void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 20.11.3.9, 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;

template<class T, class U>
    const_pointer_cast(const shared_ptr<U>& r) noexcept;

template<class T, class U>
    const_pointer_cast(shared_ptr<U>&& r) noexcept;

template<class T, class U>
    reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

template<class T, class U>
    reinterpret_pointer_cast(shared_ptr<U>&& r) noexcept;

// 20.11.3.10, shared_ptr get_deleter
template<class D, class T>
    D* get_deleter(const shared_ptr<T>& p) noexcept;

// 20.11.3.11, shared_ptr I/O
template<class E, class T, class Y>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const shared_ptr<Y>& p);

// 20.11.4, class template weak_ptr
template<class T> class weak_ptr;
// 20.11.4.6, 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>>;

// 31.8.6, 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 [pointer.traits]

The class template pointer_traits supplies a uniform interface to certain attributes of pointer-like types.

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;
};
}

20.10.3.1 Member types [pointer.traits.types]

using element_type = see below;

1 Type: Ptr::element_type if the qualified-id Ptr::element_type is valid and denotes a type (13.10.2); 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.

using difference_type = see below;

2 Type: Ptr::difference_type if the qualified-id Ptr::difference_type is valid and denotes a type (13.10.2); otherwise, ptrdiff_t.

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 (13.10.2); 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.
20.10.3.2 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 $\text{addressof}($r$)$.
4. **Remarks:** If element_type is cv void, the type of $r$ is unspecified; otherwise, it is element_type&.

20.10.3.3 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;

2. **Returns:** A pointer of type element_type* that references the same location as the argument $p$.
3. **[Note:** This function should 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.
2. **Returns:** $p$.

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 20.10.3.3), 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.

```cpp
void declare_reachable(void* $p$);
1. **Preconditions:** $p$ is a safely-derived pointer (6.7.5.4.3) or a null pointer value.
2. **Effects:** If $p$ is not null, the complete object referenced by $p$ is subsequently declared reachable (6.7.5.4.3).
3. **Throws:** May throw bad_alloc if the system cannot allocate additional memory that may be required to track objects declared reachable.

```cpp
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.
6. **Returns:** A safely derived copy of $p$ which compares equal to $p$.
7. **Throws:** Nothing.
8. **[Note:** It is expected that calls to declare_reachable($p$) will 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. Long running programs should arrange that calls are matched. — end note]

```cpp
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:** In a garbage-collecting
implementation, the fact that a region in an object is registered with `declare_no_pointers()` should 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 indirectation 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: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. —end note]

**Throws:** Nothing.

[Note: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. —end note]

```cpp
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.

```cpp
pointer_safety get_pointer_safety() noexcept;
```

**Returns:** `pointer_safety::strict` if the implementation has strict pointer safety (6.7.5.4.3). It is implementation-defined whether `get_pointer_safety()` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety.²¹⁹

### 20.10.6 Pointer alignment

```cpp
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: 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]

```cpp
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 (7.3.5) to T, where X has alignment N (6.7.6).

**Returns:** ptr.

**Throws:** Nothing.

[Note: The alignment assumption on an object X expressed by a call to `assume_aligned` may 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

```cpp
namespace std {
    struct allocator_arg_t { explicit allocator_arg_t() = default; };
}
```

²¹⁹ `pointer_safety::preferred` might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.
inline constexpr allocator_arg_t allocator_arg{};
}

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 (20.15.1). 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 (13.10.2) 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. The first argument of a constructor has type allocator_arg_t and the second argument has type Alloc or
2. The last argument of a constructor has type Alloc.

20.10.8.2 Uses-allocator construction

Uses-allocator construction with allocator 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:

1. If T does not use an allocator compatible with alloc, then alloc is ignored.
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.
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: 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]

```cpp
template<class T, class Alloc, class... Args>
constexpr auto uses_allocator_construction_args(const Alloc& alloc,
        Args&&... args) noexcept -> see below;
```

Constraints: T is not a specialization of pair.

Returns: A tuple value determined as follows:

1. If uses_allocator_v<T, Alloc> is false and is_constructible_v<T, Args...> is true, return forward_as_tuple(std::forward<Args>(args)...).
2. Otherwise, if uses_allocator_v<T, Alloc> is true and is_constructible_v<T, allocator_arg_t, const Alloc&, Args...> is true, return tuple<allocator_arg_t, const Alloc&, Args&&...>(
   allocator_arg, alloc, std::forward<Args>(args)...)
Otherwise, if `uses_allocator_v<T, Alloc>` is true and `is_constructible_v<T, Args..., const Alloc&>` is true, return `forward_as_tuple(std::forward<Args>(args)..., alloc)`.

— Otherwise, the program is ill-formed.

[Note: 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]

```cpp
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:

```cpp
defined auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct,
    Tuple1&& x, Tuple2&& y)
    noexcept -> see below;
```

Constraints: `T` is a specialization of `pair`.

Effects: Equivalent to:

```cpp
defined auto uses_allocator_construction_args(const Alloc& alloc) noexcept -> see below;
```

Constraints: `T` is a specialization of `pair`.

Effects: Equivalent to:

```cpp
defined 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:

```cpp
defined 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:

```cpp
defined 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:

```cpp
defined 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:

```cpp
defined 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:

```cpp
defined 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:

```cpp
defined auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>&& pr) noexcept -> see below;
```
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

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: 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> 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> using rebind_alloc = see below;
        template<class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;

        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);
    };
}
20.10.9.1 Member types [allocator.traits.types]

using pointer = see below;
1
   Type: Alloc::pointer if the qualified-id Alloc::pointer is valid and denotes a type (13.10.2);
   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
   (13.10.2); 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
   (13.10.2); 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 de-
   notes a type (13.10.2); 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 (13.10.2); otherwise, pointer_traits<pointer>::rebind<const void>.

using size_type = see below;
6
   Type: Alloc::size_type if the qualified-id Alloc::size_type is valid and denotes a type (13.10.2);
   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_cop-
   y_assignment is valid and denotes a type (13.10.2); 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_mov-
   e_assignment is valid and denotes a type (13.10.2); 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 (13.10.2); 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 (13.10.2); 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 (13.10.2); 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.

20.10.9.2 Static member functions [allocator.traits.members]

[[nodiscard]] static constexpr pointer allocate(Alloc& a, size_type n);
1
   Returns: a.allocate(n).

[[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).

   Throws: Nothing.
template<class T, class... Args>
static constexpr void construct(Alloc & a, T* p, Args&&... args);

Effects: Calls a.construct(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

All specializations of the default allocator meet the allocator completeness requirements (16.5.3.5.1).

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);
  };
}

20.10.10.1 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.

Remarks: The storage for the array is obtained by calling ::operator new (17.6.2), 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.

Throws: bad_array_new_length if numeric_limits<size_t>::max() / sizeof(T) < n, or bad_alloc if the storage cannot be obtained.

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 (17.6.2), but it is unspecified when this function is called.

20.10.10.2 Operators [allocator.globals]

template<class T, class U>
constexpr bool operator==(const allocator<T>&, const allocator<U>&) noexcept;

Returns: true.

20.10.11 addressof [specialized.addressof]

template<class T> constexpr T* addressof(T& r) noexcept;

Returns: The actual address of the object or function referenced by r, even in the presence of an overloaded operator&.

Remarks: An expression addressof(E) is a constant subexpression (16.3.6) if E is an lvalue constant subexpression.

20.10.12 C library memory allocation [c.malloc]

Note: The header <cstdlib> (17.2.2) declares the functions described in this subclause. — end note]

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);

Effects: These functions have the semantics specified in the C standard library.

Remarks: These functions do not attempt to allocate storage by calling ::operator new() (17.6.2).

Storage allocated directly with these functions is implicitly declared reachable (see 6.7.5.4.3) 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: This allows existing C libraries to remain unaffected by restrictions on pointers that are not safely derived, at the expense of providing far fewer garbage collection and leak detection options for malloc()-allocated objects. It also allows malloc() to be implemented with a separate allocation arena, bypassing the normal declare_reachable() implementation. The above functions should never intentionally be used as a replacement for declare_reachable(), and newly written code is strongly encouraged to treat memory allocated with these functions as though it were allocated with operator new. — end note]

These functions implicitly create objects (6.7.2) in the returned region of storage and return a pointer to a suitable created object. In the case of calloc and realloc, the objects are created before the storage is zeroed or copied, respectively.

void free(void* ptr);

Effects: This function has the semantics specified in the C standard library.

Remarks: This function does not attempt to deallocate storage by calling ::operator delete().

See also: ISO C 7.22.3

20.11 Smart pointers [smartptr]

20.11.1 Class template unique_ptr [unique.ptr]

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.

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).

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.

Each object of a type U instantiated from the unique_ptr template specified in this subclause has the strict ownership semantics, specified above, of a unique pointer. In partial satisfaction of these semantics,
each such U is Cpp17MoveConstructible and Cpp17MoveAssigneble, but is not Cpp17CopyConstructible nor Cpp17CopyAssigneble. The template parameter T of unique_ptr may be an incomplete type.

[Note: The uses of unique_ptr include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. — end note]

20.11.1 Default deleters

20.11.1.1 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 default_delete

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;
    };
}

template<class U> default_delete(const default_delete<U>& other) noexcept;

Constraints: U* is implicitly convertible to T*.

Effects: Constructs a default_delete object from another default_delete<U> object.

void operator()(T* ptr) const;

Mandates: T is a complete type.

Effects: Calls delete on ptr.

20.11.1.3 default_delete<T[]>

namespace std {
    template<class T> struct default_delete<T[]> {
        constexpr default_delete() noexcept = default;
        template<class U> default_delete(const default_delete<U[]>&) noexcept;
        template<class U> void operator()(U* ptr) const;
    };
}

template<class U> default_delete(const default_delete<U[]>& other) noexcept;

Constraints: U(*)[] is convertible to T(*)[].

Effects: Constructs a default_delete object from another default_delete<U[]> object.

template<class U> void operator()(U* ptr) const;

Mandates: U is a complete type.

Constraints: U(*)[] is convertible to T(*)[].

Effects: Calls delete[] on ptr.

20.11.1.2 unique_ptr for single objects

namespace std {
    template<class T, class D = default_delete<T>> class unique_ptr {
        using pointer = see below;
        using element_type = T;
        using deleter_type = D;
    };
}
// 20.11.1.2.1, 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;
// 20.11.1.2.2, destructor
~-unique_ptr();
// 20.11.1.2.3, 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;
// 20.11.1.2.4, 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;
// 20.11.1.2.5, 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 (13.10.2), 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: 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]

20.11.1.2.1 Constructors [unique.ptr.single_ctor]
constexpr unique_ptr() noexcept;
constexpr unique_ptr(nullptr_t) noexcept;
1 Preconditions: D meets the Cpp17DefaultConstructible requirements (Table 27), and that construction does not throw an exception.
2 Constraints: is_pointer_v<deleter_type> is false and is_default_constructible_v<deleter_type> is true.
3 Effects: Constructs a unique_ptr object that owns nothing, value-initializing the stored pointer and the stored deleter.
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 (12.4.1.8).

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 (12.4.1.8).

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:

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: The construction of the deleter can be implemented with std::forward<D>. — 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 Ivalue 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. \[ Note: The deleter constructor can be implemented with `std::forward<E>`. \ — end note \]

Postconditions: `get()` yields the value \( u.get() \) yielded before the construction. \( u.get() == \text{nullptr} \).

`get_deleter()` returns a reference to the stored deleter that was constructed from \( u.get_deleter() \).

### 20.11.1.2.2 Destructor

```cpp
~unique_ptr();
```

Preconditions: The expression `get_deleter()(get())` is well-formed, has well-defined behavior, and does not throw exceptions. \[ Note: The use of `default_delete` requires \( T \) to be a complete type. \ — end note \]

Effects: If `get() == \text{nullptr}` there are no effects. Otherwise `get_deleter()(get())`.

### 20.11.1.2.3 Assignment

```cpp
unique_ptr& operator=(unique_ptr&& u) noexcept;
```

Constraints:

1. `is_move_assignable_v<D>` is true.
2. 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())`.

Returns: `*this`.

Postconditions: \( u.get() == \text{nullptr} \).

```cpp
template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

Constraints:

1. `unique_ptr<U, E>::pointer` is implicitly convertible to `pointer`, and
2. \( U \) is not an array type, and
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())`.

Returns: `*this`.

```cpp
unique_ptr& operator=(nullptr_t) noexcept;
```

Effects: As if by `reset()`.

Returns: `*this`.

### 20.11.1.2.4 Observers

```cpp
add_lvalue_reference_t<T> operator*() const;
```

Preconditions: `get() != \text{nullptr}`.

Returns: `*get()`.
pointer operator->() const noexcept;

Preconditions: get() != nullptr.
Returns: get().

[Note: The use of this function typically requires that T be a complete type. —end note]

pointer get() const noexcept;
Returns: The stored pointer.
deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;

Returns: A reference to the stored deleter.
explicit operator bool() const noexcept;
Returns: get() != nullptr.

20.11.1.2.5 Modifiers

pointer release() noexcept;
Preconditions: get() == nullptr.
Returns: The value get() had at the start of the call to release.

void reset(pointer p = pointer()) noexcept;
Preconditions: The expression get_deleter()(get()) is well-formed, has well-defined behavior, and does not throw exceptions.
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: The order of these operations is significant because the call to get_deleter() may destroy *this. —end note]
Postconditions: get() == p. [Note: 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;
Preconditions: get_deleter() is swappable (16.5.3.2) and does not throw an exception under swap.
Effects: Invokes swap on the stored pointers and on the stored deleters of *this and u.

20.11.1.3 unique_ptr for array objects with a runtime length

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;

    // 20.11.1.3.1, 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();

    // assignment
    unique_ptr& operator=(unique_ptr&& u) noexcept;
    template<class U, class E>
        unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;

§ 20.11.1.3
unique_ptr& operator=(nullptr_t) noexcept;

// 20.11.1.3.3, observers
T& operator[](size_t i) const;
pointer get() const noexcept;
delete_type& get_deleter() noexcept;
const delete_type& get_deleter() const noexcept;
explicit operator bool() const noexcept;

// 20.11.1.3.4, 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;

1 A specialization for array types is provided with a slightly altered interface.

(1.1) — 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.

(1.2) — Pointers to types derived from T are rejected by the constructors, and by reset.

(1.3) — The observers operator* and operator-> are not provided.

(1.4) — The indexing observer operator[] is provided.

(1.5) — The default deleter will call delete[].

2 Descriptions are provided below only for members that differ from the primary template.

3 The template argument T shall be a complete type.

20.11.1.3.1 Constructors [unique.ptr.runtime.ctor]
template<class U> explicit unique_ptr(U p) noexcept;

1 This constructor behaves the same as the constructor in the primary template that takes a single parameter of type pointer.

2 Constraints:

(2.1) — U is the same type as pointer, or

(2.2) — 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.

4 Constraints:

(4.1) — U is the same type as pointer,

(4.2) — U is nullptr_t, or

(4.3) — 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;

5 This constructor behaves the same as in the primary template.

6 Constraints: Where UP is unique_ptr<U, E>:

(6.1) — U is an array type, and
20.11.1.3.2 Assignment

```cpp
template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
```

1. This operator behaves the same as in the primary template.

2. **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: This replaces the Constraints: specification of the primary template. — end note]
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.

20.11.1.5 Specialized algorithms [unique.ptr.special]

template<class T1, class D> void swap(unique_ptr<T1, D>& x, unique_ptr<T1, 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>
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: !(x < y).

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);

§ 20.11.5 638
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>
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 !(x < nullptr). The second function template returns !(nullptr < x).

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).

20.11.1.6 I/O

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

namespace std {
    class bad_weak_ptr : public exception {
    public:
        // see 17.9.2 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.
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.

```cpp
namespace std {
    template<class T> class shared_ptr {
    public:
        using element_type = remove_extent_t<T>;
        using weak_type = weak_ptr<T>;

        // 20.11.3.1, 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 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);

        // 20.11.3.2, destructor
        ~shared_ptr();

        // 20.11.3.3, 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);

        // 20.11.3.4, 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: `T` may be a function type. —end note]

[Example:

```cpp
if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```
—end example]

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[]`.

### 20.11.3.1 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:

```cpp
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).

```cpp
constexpr shared_ptr() noexcept;
```

**Postconditions:** `use_count() == 0` && `get() == nullptr`.

```cpp
template<class Y> explicit shared_ptr(Y* p);
```

**Mandates:** `Y` is a complete type.

**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*`.

**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.

**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.

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);

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.

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: To avoid the possibility of a dangling pointer, the user of this constructor should ensure that p remains valid at least until the ownership group of r is destroyed. —end note]

[Note: This constructor allows creation of an empty shared_ptr instance with a non-null stored pointer. —end note]

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().

shared_ptr(shared_ptr& r) noexcept;

template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;

Constraints: For the second constructor, Y* is compatible with T*.

Effects: Move constructs a shared_ptr instance from r.

Postconditions: *this shall contain the old value of r. r shall be empty. r.get() == nullptr.

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.
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.

20.11.3.2 Destructor

~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: 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]

20.11.3.3 Assignment

shared_ptr& operator=(const shared_ptr& r) noexcept;

Effects: Equivalent to shared_ptr(r).swap(*this).

Returns: *this.

shared_ptr& operator=(shared_ptr&& r) noexcept;

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

20.11.3.4 Modifiers

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).

20.11.3.5 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].
   
   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.
   
   Throws: Nothing.

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: get() == nullptr does not imply a specific return value of use_count(). — end note]
   
   [Note: weak_ptr<T>::lock() can affect the return value of use_count(). — end note]
   
   [Note: When multiple threads can affect the return value of use_count(), the result should be treated as 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.
20.11.3.6 Creation

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.

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.

Returns: A shared_ptr instance that stores and owns the address of the newly constructed object.

Postconditions: r.get() != 0 && r.use_count() == 1, where r is the return value.

Throws: bad_alloc, or an exception thrown from allocate or from the initialization of the object.

Remarks:

(7.1) — Implementations should perform no more than one memory allocation. [Note: This provides efficiency equivalent to an intrusive smart pointer. — end note]

(7.2) — 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.

(7.3) — 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.

(7.4) — 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.

(7.5) — 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

(7.5.1) allocator_traits<A2>::construct(a2, pv, v) or

(7.5.2) 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>.

(7.6) — 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.

(7.7) — 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>.

(7.8) — 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.

(7.9) — 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 `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 `allocate_shared` is to be destroyed, it is destroyed via the expression `allocator_traits<A2>::destroy(a2, pv)` where `pv` points to that object of type `remove_cv_t<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>`.

[Note: These functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as reference counts. — end note]

template<class T, class... Args>
shared_ptr<T> make_shared(Args&&... args); // T is not array

template<class T, T class A, class... Args>
shared_ptr<T> allocate_shared(const A& a, Args&&... args); // T is not array

Constraints: $T$ is not an array type.

Returns: A `shared_ptr` to an object of type $T$ with an initial value $T$(forward<Args>(args)...).

Remarks: The `shared_ptr` constructors called by these functions enable `shared_from_this` with the address of the newly constructed object of type $T$.

[Example:

code

shared_ptr<int> p = make_shared<int>(); // shared_ptr to int()
shared_ptr<vector<int>> q = make_shared<vector<int>>\((16, 1)\);
// shared_ptr to vector of 16 elements with value 1

— end example]

template<class 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[]

Constraints: $T$ is of the form $U[]$.

Returns: A `shared_ptr` to an object of type $U[N]$ with a default initial value, where $U$ is `remove_extent_t<T>`.  

[Example:

code

shared_ptr<double[]> p = make_shared<double[]>\((1024)\);
// shared_ptr to a value-initialized double[1024]
shared_ptr<double[]> q = make_shared<double[]>\((6)\);
// shared_ptr to a value-initialized double[6]

— end example]

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]

Constraints: $T$ is of the form $U[N]$.

Returns: A `shared_ptr` to an object of type $T$ with a default initial value.

[Example:

code

shared_ptr<double[1024]> p = make_shared<double[1024]>();
// shared_ptr to a value-initialized double[1024]
shared_ptr<double[]> q = make_shared<double[]>();
// shared_ptr to a value-initialized double[6]

— end example]
template<class T>
    shared_ptr<T> make_shared(size_t N,
    const remove_extent_t<T>& u);     // T is U[N]

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:
    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> q = make_shared<double[2]>(6, {1.0, 0.0});
    // shared_ptr to a double[6][2], where each double[2] element is {1.0, 0.0}
    shared_ptr<vector<int[]>> r = make_shared<vector<int[]>>(4, {1, 2});
    // 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, 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 T, where each array element of type remove_extent_t<T> has an initial value of u.

[Example:
    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<vector<int>> r = make_shared<vector<int>>(4, {1, 2});
    // 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:
    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> 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>. 647
Example:
```cpp
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
```

## 20.11.3.7 Comparison

### template<class T, class U>
```cpp
bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

Returns: `a.get() == b.get()`.

### template<class T>
```cpp
bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
```

Returns: `!a`.

### template<class T, class U>
```cpp
strong_ordering operator<=>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
```

Returns: `compare_three_way()(a.get(), b.get())`.

### Note: Defining a comparison function allows `shared_ptr` objects to be used as keys in associative containers.

### template<class T>
```cpp
strong_ordering operator<=>(const shared_ptr<T>& a, nullptr_t) noexcept;
```

Returns: `compare_three_way()(a.get(), nullptr)`.

## 20.11.3.8 Specialized algorithms

### template<class T>
```cpp
void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;
```

Effects: Equivalent to `a.swap(b)`.

## 20.11.3.9 Casts

### template<class T, class U>
```cpp
shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
```

### template<class T, class U>
```cpp
shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
```

Mandates: The expression `static_cast<T*>((U*)nullptr)` is well-formed.

Returns:
- `shared_ptr<T>(R, static_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: The seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice.

### template<class T, class U>
```cpp
shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
```

### template<class T, class U>
```cpp
shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
```

Mandates: The expression `dynamic_cast<T*>(nullptr)` is well-formed. The expression `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` is well formed.

Preconditions: The expression `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` has well-defined behavior.

Returns:
- When `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` returns a non-null value `p`, `shared_ptr<T>(R, p)`, where `R` is `r` for the first overload, and `std::move(r)` for the second.
- Otherwise, `shared_ptr<T>()`. 

§ 20.11.3.9
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 \texttt{const\_cast<T*>(U*)nullptr)} is well-formed.

Returns:
\texttt{shared\_ptr<T>(R, \texttt{const\_cast\_typename\_shared\_ptr<T>::element\_type*}}(r.get()))
where R is r for the first overload, and \texttt{std::move}(r) for the second.

Note: The seemingly equivalent expression \texttt{shared\_ptr<T>(\texttt{const\_cast<T*>(U*)nullptr)}} 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 \texttt{reinterpret\_cast<T*>(U*)nullptr)} is well-formed.

Returns:
\texttt{shared\_ptr<T>(R, \texttt{reinterpret\_cast\_typename\_shared\_ptr<T>::element\_type*}}(r.get()))
where R is r for the first overload, and \texttt{std::move}(r) for the second.

Note: The seemingly equivalent expression \texttt{shared\_ptr<T>(\texttt{reinterpret\_cast<T*>(U*)nullptr)}} will eventually result in undefined behavior, attempting to delete the same object twice. — end note

§ 20.11.4
template<class Y>
    weak_ptr(weak_ptr<Y>&& r) noexcept;

// 20.11.4.2, destructor
~weak_ptr();

// 20.11.4.3, assignment
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;

// 20.11.4.4, modifiers
void swap(weak_ptr& r) noexcept;
void reset() noexcept;

// 20.11.4.5, observers
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;
};

template<class T>
weak_ptr(shared_ptr<T>) -> weak_ptr<T>;

// 20.11.4.6, specialized algorithms
template<class T>
void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

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.

20.11.4.1 Constructors [util.smartptr.weak.const]

constexpr weak_ptr() noexcept;
1
   Effects: Constructs an empty weak_ptr object.
2
   Postconditions: use_count() == 0.

weak_ptr(const weak_ptr& r) noexcept;
template<class Y> weak_ptr(const weak_ptr<Y>& r) noexcept;
template<class Y>
    weak_ptr(const shared_ptr<Y>& r) noexcept;
3
   Constraints: For the second and third constructors, Y* is compatible with T*.
4
   Effects: If r is empty, constructs an empty weak_ptr object; otherwise, constructs a weak_ptr object
   that shares ownership with r and stores a copy of the pointer stored in r.
5
   Postconditions: use_count() == r.use_count().

weak_ptr(weak_ptr&& r) noexcept;
template<class Y>
    weak_ptr(weak_ptr<Y>&& r) noexcept;
6
   Constraints: For the second constructor, Y* is compatible with T*.
7
   Effects: Move constructs a weak_ptr instance from r.
8
   Postconditions: *this shall contain the old value of r. r shall be empty. r.use_count() == 0.
20.11.4.2 Destructor

~weak_ptr();
1 *Effects:* Destroys this weak_ptr object but has no effect on the object its stored pointer points to.

20.11.4.3 Assignment

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;
1 *Effects:* Equivalent to weak_ptr(r).swap(*this).
2 *Remarks:* The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary object.
3 *Returns:* *this.

weak_ptr& operator=(weak_ptr&& r) noexcept;
template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;
4 *Effects:* Equivalent to weak_ptr(std::move(r)).swap(*this).
5 *Returns:* *this.

20.11.4.4 Modifiers

void swap(weak_ptr& r) noexcept;
1 *Effects:* Exchanges the contents of *this and r.
void reset() noexcept;
2 *Effects:* Equivalent to weak_ptr().swap(*this).

20.11.4.5 Observers

long use_count() const noexcept;
1 *Returns:* 0 if *this is empty; otherwise, the number of shared_ptr instances that share ownership with *this.

bool expired() const noexcept;
2 *Returns:* use_count() == 0.

shared_ptr<T> lock() const noexcept;
3 *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;
4 *Returns:* An unspecified value such that

— x.owner_before(y) defines a strict weak ordering as defined in 25.8;
— 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.

20.11.4.6 Specialized algorithms

template<class T>
void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;
1 *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;

§ 20.11.5
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: Note that

(2.1) — operator() defines a strict weak ordering as defined in 25.8;
(2.2) — under the equivalence relation defined by operator(), !operator()(a, b) && !operator()(b, a),
    two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both
    empty.

— end note]

20.11.6 Class template enable_shared_from_this

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:
    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&);
            ~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;

§ 20.11.6

652
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: 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

Letting UP be unique_ptr<T,D>, the specialization hash<UP> is enabled (20.14.18) 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.

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

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> 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` [mem.res.class]

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;
    }
}
```

### 20.12.2.1 Public member functions [mem.res.public]

- `~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);

### 20.12.2.2 Private virtual member functions [mem.res.private]

- `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 (6.7.5.4.1) 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;

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: 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]

20.12.2.3 Equality

bool operator==(const memory_resource& a, const memory_resource& b) noexcept;

Returns: &a == &b || a.is_equal(b).

20.12.3 Class template polymorphic_allocator

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.

All specializations of class template pmr::polymorphic_allocator meet the allocator completeness requirements (16.5.3.5.1).

namespace std::pmr {
  template<class Tp = byte> class polymorphic_allocator {
    memory_resource* memory_rsrc; // exposition only

    public:
      using value_type = Tp;

      // 20.12.3.1, 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;

      // 20.12.3.2, 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;
  };
}
20.12.3.1 Constructors

polymorphic_allocator() noexcept;

   Effects: Sets memory_rsrc to get_default_resource().

polymorphic_allocator(memory_resource* r);

   Preconditions: r is non-null.
   Effects: Sets memory_rsrc to r.
   Throws: Nothing.
   [Note: This constructor provides an implicit conversion from memory_resource*. — end note]

template<class U> polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

6
   Effects: Sets memory_rsrc to other.resource().

20.12.3.2 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: The return type is void* (rather than, e.g., byte*) to support conversion to an arbitrary pointer type U* by static_cast<U*>(), 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));

7
   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)));

9
   [Note: 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>();

§ 20.12.3.2 656
try {
    construct(p, std::forward<CtorArgs>(ctor_args)...);
} catch (...) {
    deallocate_object(p);
    throw;
}
return p;

[Note: 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);
delealect_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: The memory resource is not propagated. — end note]

memory_resource* resource() const;

Returns: memory_rsrc.

20.12.3.3 Equality

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

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().

§ 20.12.4

657
memory_resource* set_default_resource(memory_resource* r) noexcept;
4 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().
5 Returns: The previous value of the default memory resource pointer.
6 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.

memory_resource* get_default_resource() noexcept;
7 Returns: The current value of the default memory resource pointer.

20.12.5 Pool resource classes [mem.res.pool]
20.12.5.1 Classes synchronized_pool_resource and unsynchronized_pool_resource [mem.res.pool.overview]
1 The synchronized_pool_resource and unsynchronized_pool_resource classes (collectively called pool
resource classes) are general-purpose memory resources having the following qualities:

(1.1) — Each resource frees its allocated memory on destruction, even if deallocate has not been called for
some of the allocated blocks.
(1.2) — 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.
(1.3) — 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:
By allocating memory in chunks, the pooling strategy increases the chance that consecutive allocations
will be close together in memory. — end note]
(1.4) — Allocation requests that exceed the largest block size of any pool are fulfilled directly from the upstream
allocator.
(1.5) — A pool_options struct may be passed to the pool resource constructors to tune the largest block size
and the maximum chunk size.

2 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.

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()) {}  // default
        explicit synchronized_pool_resource(memory_resource* upstream)
            : synchronized_pool_resource(pool_options(), upstream) {}  // explicit
        explicit synchronized_pool_resource(const pool_options& opts)
            : synchronized_pool_resource(opts, get_default_resource()) {}  // explicit
        synchronized_pool_resource(const synchronized_pool_resource&) = delete;
        virtual ~synchronized_pool_resource();

        synchronized_pool_resource& operator=(const 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)
    : unsynchronized_pool_resource(pool_options(), 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: 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().

Effects: Calls release().

20.12.5.4 Members

virtual =synchronized_pool_resource();
virtual =unsynchronized_pool_resource();
Effects: Calls release().

virtual ~synchronized_pool_resource();
virtual ~unsynchronized_pool_resource();

Effects: Calls release().

memory_resource* upstream_resource() const;

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;

Returns: A pointer to allocated storage (6.7.5.4.1) 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).

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().

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

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.
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 `deallocate` has not been called for some of the allocated blocks.

```cpp
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;
    }
}
```

### 20.12.6.1 Constructors and destructor

```cpp
explicit monotonic_buffer_resource(memory_resource* upstream);
monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);
```

1. **Preconditions:** `upstream` is the address of a valid memory resource. `initial_size`, if specified, is greater than zero.
2. **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.

```cpp
monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream);
```

3. **Preconditions:** `upstream` is the address of a valid memory resource. `buffer_size` is no larger than the number of bytes in `buffer`.
4. **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).

```cpp
~monotonic_buffer_resource();
```
5. **Effects:** Calls `release()`.
20.12.6.2 Members

void release();

Effects: Calls upstream_rsrc->deallocate() as necessary to release all allocated memory.

[Note: 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.

do_allocate(size_t bytes, size_t alignment) override;

Returns: A pointer to allocated storage (6.7.5.4.1) 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).

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.

Throws: Nothing unless upstream_rsrc->allocate() throws.

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.

do_is_equal(const memory_resource& other) const noexcept override;

Returns: this == &other.

20.13 Class template scoped_allocator_adaptor

20.13.1 Header <scoped_allocator> synopsis

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: 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]

namespace std {
    template<class OuterAlloc, class... InnerAllocs>
    class scoped_allocator_adaptor {
        public:  // 20.13.1 662
            template<class OuterAlloc1, class OuterAlloc2, class... InnerAllocs>
            bool operator==(const scoped_allocator_adaptor<OuterAlloc1, InnerAllocs...>& a,
                            const scoped_allocator_adaptor<OuterAlloc2, InnerAllocs...>& b) noexcept;
    }
using OuterTraits = allocator_traits<OuterAlloc>;// exposition only
scoped_allocator_adaptor<InnerAllocs...> inner;// exposition only

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<OuterA2, InnerAllocs...>& other) noexcept;
scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;

template<class OuterA2>
    scoped_allocator_adaptor(
        const 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<class T, class... Args>
    void construct(T* p, Args&&... args);

template<class 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;
1
    Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise,
    scoped_allocator_adaptor<InnerAllocs...>.

using propagate_on_container_copy_assignment = see below;
2
    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;
3
    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;
4
    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;
5
    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();
1
    Effects: Value-initializes the OuterAlloc base class and the inner allocator object.

template<class OuterA2>
scoped_allocator_adaptor(OuterA2&& outerAlloc, const InnerAllocs&... innerAllocs) noexcept;
2
    Constraints: is_constructible_v<OuterAlloc, OuterA2> is true.
3
    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;
4
    Effects: Initializes each allocator within the adaptor with the corresponding allocator from other.

scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
5
    Effects: Move constructs each allocator within the adaptor with the corresponding allocator rvalue from
other.

template<class OuterA2>
scoped_allocator_adaptor(
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
6
    Constraints: is_constructible_v<OuterAlloc, OuterA2> is true.
7
    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;
8
    Constraints: is_constructible_v<OuterAlloc, OuterA2> is true.
9
    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 (13.10.2) and x otherwise; OUTERMOST_ALLOC_TRAITS(x) is allocator_traits<remove_reference_t<decltype(OUTERMOST(x))>>::. [Note: 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]

inner_allocator_type& inner_allocator() noexcept;
const inner_allocator_type& inner_allocator() const noexcept;

Returns: *this if sizeof...(InnerAllocs) is zero; otherwise, inner.

outer_allocator_type& outer_allocator() noexcept;
Returns: static_cast<OuterAlloc&>(*this).

const outer_allocator_type& outer_allocator() const noexcept;
Returns: static_cast<const OuterAlloc&>(*this).

[[nodiscard]] pointer allocate(size_type n);
Returns: allocator_traits<OuterAlloc>::allocate(outer_allocator(), n).

[[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);
Returns: allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint).

void deallocate(pointer p, size_type n) noexcept;
Effects: As if by: allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n);

size_type max_size() const;
Returns: allocator_traits<OuterAlloc>::max_size(outer_allocator()).

template<class T, class... Args>
void construct(T* p, Args&&... args);
Effects: Equivalent to:
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)...));

template<class T>
void destroy(T* p);
Effects: Calls OUTERMOST_ALLOC_TRAITS(*this)::destroy(OUTERMOST(*this), p).

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

template<class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a, const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;

Returns: If sizeof...(InnerAllocs) is zero,
   a.outer_allocator() == b.outer_allocator()
otherwise
A function object type is an object type (6.8) that can be the type of the postfix-expression in a function call (7.6.1.2, 12.4.1.1). 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 Function objects

A function object type is an object type (6.8) that can be the type of the postfix-expression in a function call (7.6.1.2, 12.4.1.1). 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.

20.14.1 Header <functional> synopsis

namespace std {
    // 20.14.4, invoke
    template<class F, class... Args>
    constexpr invoke_result_t<F, Args...> invoke(F&& f, Args&&... args) noexcept(is_nothrow_invocable_v<F, Args...>);

    // 20.14.5, 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;

    // 20.14.6, 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>;

    // 20.14.7, 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>;

    // 20.14.7.7, class compare_three_way
    struct compare_three_way;

220) 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.
// 20.14.9, 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>;

// 20.14.10, 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>;

// 20.14.11, identity
struct identity;

// 20.14.12, function template not_fn
template<class F> constexpr unspecified not_fn(F&& f);

// 20.14.13, function template bind_front
template<class F, class... Args> constexpr unspecified bind_front(F&&, Args&&...);

// 20.14.14, 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;
}

// 20.14.15, member function adaptors
template<class R, class T>
constexpr unspecified mem_fn(R T::* ) noexcept;

// 20.14.16, polymorphic function wrappers
class bad_function_call;

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;

// 20.14.17, 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;

// 20.14.18, class template hash
template<class T>
    struct hash;

namespace ranges {
    // 20.14.8, concept-constrained comparisons
    struct equal_to;
    struct not_equal_to;
    struct greater;
    struct less;
    struct greater_equal;
    struct less_equal;
}
}

[Example: 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: To negate every element of a:
    transform(a.begin(), a.end(), a.begin(), negate<double>());
    —end example]

20.14.2 Definitions [func.def]

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.

20.14.3 Requirements [func.require]

1 Define \textit{INVOLVE}(f, t_1, t_2, \ldots, t_N) as follows:
§ 20.14.4 Function template `invoke`  

```
template<class F, class... Args>
    constexpr invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
      noexcept(is_nothrow_invocable_v<F, Args...>);
```

Returns: `INVOKE(std::forward<F>(f), std::forward<Args>(args)...)` (20.14.3).

20.14.5 Class template `reference_wrapper`  

```
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;

reference_wrapper<T> is a Cpp17CopyConstructible and Cpp17CopyAssignable wrapper around a reference to an object or function of type T.

reference_wrapper<T> is a trivially copyable type (6.8).
The template parameter T of reference_wrapper may be an incomplete type.

20.14.5.1 Constructors and destructor

Let FUN denote the exposition-only functions

void FUN(T&) noexcept;
void FUN(T&&) = delete;

Constraints: The expression FUN(declval<U>()) is well-formed and is_same_v<remove_cvref_t<U>, reference_wrapper> is false.

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.

Remarks: The expression inside noexcept is equivalent to noexcept(FUN(declval<U>())).

20.14.5.2 Assignment

Postconditions: *this stores a reference to x.get().

20.14.5.3 Access

Returns: The stored reference.

20.14.5.4 Invocation

template<class... ArgTypes>
constexpr invoke_result_t<T&, ArgTypes...>
operator() (ArgTypes&&... args) const;

Mandates: T is a complete type.

Returns: INVOKE(get(), std::forward<ArgTypes>(args)...). (20.14.3)

20.14.5.5 Helper functions [refwrap.helpers]

The template parameter T of the following ref and cref function templates may be an incomplete type.

template<class T> constexpr reference_wrapper<T> ref(T& t) noexcept;
Returns: reference_wrapper<T>(t).

template<class T> constexpr reference_wrapper<T> ref(reference_wrapper<T> t) noexcept;
Returns: ref(t.get()).

template<class T> constexpr reference_wrapper<const T> cref(const T& t) noexcept;
Returns: reference_wrapper<const T>(t).

template<class T> constexpr reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;
Returns: cref(t.get()).

20.14.6 Arithmetic operations [arithmetic.operations]

The library provides basic function object classes for all of the arithmetic operators in the language (7.6.5, 7.6.6).

20.14.6.1 Class template plus [arithmetic.operations.plus]

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;
Returns: x + y.

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));
    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));
Returns: std::forward<T>(t) + std::forward<U>(u).

20.14.6.2 Class template minus [arithmetic.operations.minus]

template<class T = void> struct minus {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;
Returns: x - y.

template<> struct minus<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));
Returns: std::forward<T>(t) - std::forward<U>(u).
20.14.6.3 Class template multiplies

```
template<class T = void> struct multiplies {
    constexpr T operator()(const T& x, const T& y) const;
};
```

1 Returns: \( x \times y \).

```
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;
};
```

2 Returns: \( \text{std::forward}(t) \times \text{std::forward}(u) \).

20.14.6.4 Class template divides

```
template<class T = void> struct divides {
    constexpr T operator()(const T& x, const T& y) const;
};
```

1 Returns: \( \frac{x}{y} \).

```
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;
};
```

2 Returns: \( \text{std::forward}(t) / \text{std::forward}(u) \).

20.14.6.5 Class template modulus

```
template<class T = void> struct modulus {
    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;
};
```

2 Returns: \( \text{std::forward}(t) \% \text{std::forward}(u) \).

20.14.6.6 Class template negate

```
template<class T = void> struct negate {
    constexpr T operator()(const T& x) const;
};
```

§ 20.14.6.6
constexpr T operator()(const T& x) const;

"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));

"Returns: \(-\text{std::forward}(t)\)."
```

```
20.14.7 Comparisons

The library provides basic function object classes for all of the comparison operators in the language (7.6.9, 7.6.10).

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 (16.3.13). [Note: If \(a < b\) is well-defined for pointers \(a\) and \(b\) of type \(P\), then \((a < b) == \text{less}(P)(a, b)\), \((a > b) == \text{greater}(P)(a, b)\), 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.

```
20.14.7.1 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;

"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));

"Returns: \(\text{std::forward}(t) == \text{std::forward}(u)\)."
```

```
20.14.7.2 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));

"Returns: \(\text{std::forward}(t) != \text{std::forward}(u)\)."
```

§ 20.14.7.2
template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
  -> decltype(std::forward<T>(t) != std::forward<U>(u));

Returns: std::forward<T>(t) != std::forward<U>(u).

20.14.7.3 Class template greater [comparisons.greater]

```cpp
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;
1 Returns: x > y.
```

```cpp
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;
};
```

2 Returns: std::forward<T>(t) > std::forward<U>(u).

20.14.7.4 Class template less [comparisons.less]

```cpp
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;
1 Returns: x < y.
```

```cpp
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;
};
```

2 Returns: std::forward<T>(t) < std::forward<U>(u).

20.14.7.5 Class template greater_equal [comparisons.greater.equal]

```cpp
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;
1 Returns: x >= y.
```

```cpp
template<> struct greater_equal<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 Returns: std::forward<T>(t) >= std::forward<U>(u).

§ 20.14.7.5 674
20.14.7.6 Class template less_equal

```cpp
template<class T = void> struct less_equal {
  constexpr bool operator()(const T& x, const T& y) const;
};
```

`Returns: x <= y.`

```cpp
template<> struct less_equal<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;
};
```

`Returns: std::forward<T>(t) <= std::forward<U>(u).`

20.14.7.7 Class compare_three_way

```cpp
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;
};
```

`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 of type P, returns strong_ordering::less if (the converted value of) t precedes u in the implementation-defined strict total order over pointers (16.3.13), 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);`

20.14.8 Concept-constrained comparisons

```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;
}  
```

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.

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;
};

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: returns `false` if either (the converted value of) `t` precedes `u` or `u` precedes `t` in the implementation-defined strict total order over pointers (16.3.13) and otherwise `true`.

(3.2) — Otherwise, equivalent to: `return std::forward<T>(t) == std::forward<U>(u);`

4 `struct ranges::not_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;
};

5 `operator()` has effects equivalent to:

    return !ranges::equal_to{}(std::forward<T>(t), std::forward<U>(u));

6 `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;
};

7 `operator()` has effects equivalent to:

    return ranges::less{}(std::forward<U>(u), std::forward<T>(t));

8 `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;
};

9 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). For any expressions `ET` and `EU` such that `decltype((ET))` is `T` and `decltype((EU))` is `U`, exactly one of `ranges::less{}(ET, EU)`, `ranges::less{}(EU, ET)`, or `ranges::equal_to{}(ET, EU)` is `true`.

10 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 (16.3.13) and otherwise `false`.

(7.2) — Otherwise, equivalent to: `return std::forward<T>(t) < std::forward<U>(u);`

11 `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;
};

12 `operator()` has effects equivalent to:
return !ranges::less{}(std::forward<T>(t), std::forward<U>(u));

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;
};

operator() has effects equivalent to:
    return !ranges::less{}(std::forward<U>(u), std::forward<T>(t));

20.14.9 Logical operations

The library provides basic function object classes for all of the logical operators in the language (7.6.14, 7.6.15, 7.6.2.1).

20.14.9.1 Class template logical_and

    template<class T = void> struct logical_and {
        constexpr bool operator()(const T& x, const T& y) const;
    };  

    constexpr bool operator()(const T& x, const T& y) const;

    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;
    };  

    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) && std::forward<U>(u));

    Returns: x && y.

    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;
    };  

    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) & std::forward<U>(u));

    Returns: std::forward<T>(t) & std::forward<U>(u).

20.14.9.2 Class template logical_or

    template<class T = void> struct logical_or {
        constexpr bool operator()(const T& x, const T& y) const;
    };  

    constexpr bool operator()(const T& x, const T& y) const;

    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));

        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));

    Returns: std::forward<T>(t) || std::forward<U>(u).

20.14.9.3 Class template logical_not

    template<class T = void> struct logical_not {
        constexpr bool operator()(const T& x) const;
    };
constexpr bool operator()(const T& x) const;

Returns: !x.

template<> struct logical_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&& t) const
-> decltype(!std::forward<T>(t));

Returns: !std::forward<T>(t).

20.14.10 Bitwise operations

The library provides basic function object classes for all of the bitwise operators in the language (7.6.11, 7.6.13, 7.6.12, 7.6.2.1).

20.14.10.1 Class template bit_and

template<class T = void> struct bit_and {
    constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;

Returns: x & y.

template<> struct bit_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;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) & std::forward<U>(u));

Returns: std::forward<T>(t) & std::forward<U>(u).

20.14.10.2 Class template bit_or

template<class T = void> struct bit_or {
    constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;

Returns: x | y.

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));

Returns: std::forward<T>(t) | std::forward<U>(u).

20.14.10.3 Class template bit_xor

template<class 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).

20.14.10.4 Class template bit_not
    [bitwise.operations.not]

template<class 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).

20.14.11 Class identity
    [func.identity]

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
    Effects: Equivalent to: return std::forward<T>(t);

20.14.12 Function template not_fn
    [func.not.fn]

template<class F> constexpr unspecified not_fn(F&& f);
1
    In the text that follows:

(1.1)  — g is a value of the result of a not_fn invocation,
(1.2)  — FD is the type decay_t<F>,
(1.3)  — fd is the target object of g (20.14.2) of type FD, direct-non-list-initialized with std::forward<F
        >(f),
(1.4)  — call_args is an argument pack used in a function call expression (7.6.1.2) 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...).
Throws: Any exception thrown by the initialization of \( \text{fd} \).

### 20.14.13 Function template bind_front

```cpp
template<class F, class... Args> constexpr unspecified bind_front(F&& f, Args&&... args);
```

In the text that follows:

1. \( g \) is a value of the result of a `bind_front` invocation,
2. \( \text{FD} \) is the type `decay_t<F>`,
3. \( \text{fd} \) is the target object of \( g \) (20.14.2) of type \( \text{FD} \), direct-non-list-initialized with `std::forward<F>(f)`,
4. \( \text{BoundArgs} \) is a pack that denotes `decay_t<Args>...`,
5. \( \text{bound_args} \) is a pack of bound argument entities of \( g \) (20.14.2) of types `BoundArgs...`, direct-non-list-initialized with `std::forward<Args>(args)...`, respectively, and
6. \( \text{call_args} \) is an argument pack used in a function call expression (7.6.1.2) of \( g \).

Mandates:

\[
\text{is_constructible_v<FD, F> && is_move_constructible_v<FD> && (is_constructible_v<BoundArgs, Args> && ...) && (is_move_constructible_v<BoundArgs> && ...)}
\]

is true.

Preconditions: \( \text{FD} \) meets the Cpp17MoveConstructible requirements. For each \( T_i \) in \( \text{BoundArgs} \), if \( T_i \) is an object type, \( T_i \) meets the Cpp17MoveConstructible requirements.

Returns: A perfect forwarding call wrapper \( g \) with call pattern `invoke(fd, bound_args..., call_args...)`.

Throws: Any exception thrown by the initialization of the state entities of \( g \) (20.14.2).

### 20.14.14 Function object binders

This subclause describes a uniform mechanism for binding arguments of callable objects.

#### 20.14.14.1 Class template is_bind_expression

```cpp
namespace std {
    template<class T> struct is_bind_expression; // see below
}
```

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.

Specializations of the `is_bind_expression` template shall meet the Cpp17UnaryTypeTrait requirements (20.15.1). 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 `false_type`. A program may specialize this template for a program-defined type \( T \) to have a base characteristic of `true_type` to indicate that \( T \) should be treated as a subexpression in a `bind` call.

#### 20.14.14.2 Class template is_placeholder

```cpp
namespace std {
    template<class T> struct is_placeholder; // see below
}
```

The class template `is_placeholder` can be used to detect the standard placeholders \_1, \_2, and so on. The function template `bind` uses `is_placeholder` to detect placeholders.

Specializations of the `is_placeholder` template shall meet the Cpp17UnaryTypeTrait requirements (20.15.1). The implementation provides a definition that has the base characteristic of `integral_constant<int, J>` if \( T \) is the type of `std::placeholders::_J`, otherwise it has a base characteristic of `integral_constant<int, 0>`. A program may specialize this template for a program-defined type \( T \) to have a base characteristic of `integral_constant<int, N>` with \( N > 0 \) to indicate that \( T \) should be treated as a placeholder type.
### 20.14.14.3 Function template bind

In the text that follows:

- **g** is a value of the result of a **bind** invocation,
- **FD** is the type `decay_t<F>`,
- **fd** is an lvalue that is a target object of **g** (20.14.2) of type FD direct-non-list-initialized with `std::forward<F>(f)`,
- **T_i** is the i\(^{th}\) type in the template parameter pack `BoundArgs`,
- **TD_i** is the type `decay_t<T_i>`,
- **t_i** is the i\(^{th}\) argument in the function parameter pack `bound_args`,
- **td_i** is a bound argument entity of **g** (20.14.2) of type TD_i direct-non-list-initialized with `std::forward<T_i>(t_i)`,
- **U_j** is the j\(^{th}\) deduced type of the `UnBoundArgs&&`... parameter of the argument forwarding call wrapper, and
- **u_j** is the j\(^{th}\) argument associated with U_j.

```cpp
template<class F, class... BoundArgs>
constexpr unspecified bind(F&& f, BoundArgs&&... bound_args);

template<class R, class F, class... BoundArgs>
constexpr unspecified bind(R&& r, F&& f, BoundArgs&&... bound_args);
```

#### Mandates:
- `is_constructible_v<FD, F>` is true. For each T_i in BoundArgs, `is_constructible_v<TD_i, T_i>` is true.

#### Preconditions:
- FD and each TD_i meet the Cpp17MoveConstructible and Cpp17Destructible requirements.
- `INVOKEd(fd, w_1, w_2, ..., w_N)` (20.14.3) is a valid expression for some values w_1, w_2, ..., w_N, where N has the value `sizeof...(bound_args)`.

#### Returns:
- An argument forwarding call wrapper **g** (20.14.3). A program that attempts to invoke a volatile-qualified **g** is ill-formed. When **g** is not volatile-qualified, invocation of `g(u_1, u_2, ..., u_M)` is expression-equivalent (16.3.11) to

```cpp
INVOKEx(static_cast<INVOKE<...>(v_{td})>,
    static_cast<INVOKE<...>>(v_1), static_cast<INVOKE<...>>(v_2), ..., static_cast<INVOKE<...>>(v_N))
```

for the first overload, and

```cpp
INVOKExR(static_cast<INVOKE<...>>(v_{td}),
    static_cast<INVOKE<...>>(v_1), static_cast<INVOKE<...>>(v_2), ..., static_cast<INVOKE<...>>(v_N))
```

for the second overload, where the values and types of the target argument v_{td} and of the bound arguments v_1, v_2, ..., v_N are determined as specified below.

#### Throws:
Any exception thrown by the initialization of the state entities of **g**.

#### [Note:]
If all of FD and TD_i meet the requirements of Cpp17CopyConstructible, then the return type meets the requirements of Cpp17CopyConstructible. — end note]

The values of the **bound arguments** v_1, v_2, ..., v_N and their corresponding types V_1, V_2, ..., V_N depend on the types TD_i derived from the call to **bind** and the cv-qualifiers cv of the call wrapper **g** as follows:

- If TD_i is reference_wrapper<T>, the argument is `td_i.get()` and its type V_i is T;
- If the value of `is_bind_expression_v<TD_i>` is true, the argument is `static_cast<cv TD_i&>(td_i)(std::forward<U_j>(u_j)...)` and its type V_i is `invoke_result_t<cv TD_i&, U_j...&>`;
- If the value j of `is_placeholder_v<TD_i>` is not zero, the argument is `std::forward<U_j>(u_j)` and its type V_i is U_j&;
- Otherwise, the value is td_i and its type V_i is cv TD_i&.

The value of the target argument v_{td} is fd and its corresponding type V_{td} is cv FD&.

### 20.14.14.4 Placeholders

```cpp
namespace std::placeholders {

§ 20.14.14.4 681
```
// M is the implementation-defined number of placeholders
see below _1;
see below _2;
. .
. see below _M;
}

1 All placeholder types meet the Cpp17DefaultConstructible and 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 Cpp17CopyAssignable requirements, but if
so, their copy assignment operators are constexpr functions that do not throw exceptions.

2 Placeholders should be defined as:

    inline constexpr unspecified _1{};

If they are not, they are declared as:

    extern unspecified _1;

20.14.15 Function template mem_fn

    template<class R, class T>
    constexpr unspecified mem_fn(R T::* pm) noexcept;

1 Returns: A simple call wrapper (20.14.2) fn with call pattern invoke(pmd, call_args...), where
    pmd is the target object of fn of type R T::* direct-non-list-initialized with pm, and call_args is an
    argument pack used in a function call expression (7.6.1.2) of pm.

20.14.16 Polymorphic function wrappers

1 This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

20.14.16.1 Class bad_function_call

    namespace std {
    class bad_function_call : public exception {
        public:
            // see 17.9.2 for the specification of the special member functions
            const char* what() const noexcept override;
    }
    }

20.14.16.2 Class template function

    namespace std {
    template<class> class function; // not defined

    template<class R, class... ArgTypes>
    class function<R(ArgTypes...)> {
        public:
            using result_type = R;

            // 20.14.16.2.1, construct/copy/destroy
            function() noexcept;
            function(nullptr_t) noexcept;
            function(const function&) noexcept;
            function(function&&) noexcept;
            template<class F> function(F);

20.14.16.2 682
function& operator=(const function&);  
function& operator=(function&);  
function& operator=(nullptr_t) noexcept;  
template<class F> function& operator=(F&);  
template<class F> function& operator=(reference_wrapper<F>) noexcept;  

~function();  
// 20.14.16.2.2, function modifiers  
void swap(function&) noexcept;  
// 20.14.16.2.3, function capacity  
explicit operator bool() const noexcept;  
// 20.14.16.2.4, function invocation  
R operator()(ArgTypes...) const;  
// 20.14.16.2.5, 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>;  
// 20.14.16.2.6, null pointer comparison functions  
template<class R, class... ArgTypes>  
bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;  
// 20.14.16.2.7, specialized algorithms  
template<class R, class... ArgTypes>  
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;  
}

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 (20.14.2), given a call signature (20.14.2), allowing functions to be first-class objects.
2 A callable type (20.14.2) 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 (20.14.3).
3 The function class template is a call wrapper (20.14.2) whose call signature (20.14.2) is R(ArgTypes...).
4 [Note: The types deduced by the deduction guides for function may change in future versions of this International Standard. — end note]

20.14.16.2.1 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.

[Note: 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. —end note]

```
function(function&& f) noexcept;
```

5

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.

[Note: 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. —end note]

```
template<class F> function(F f);
```

7

Constraints: \( F \) is Lvalue-Callable (20.14.16.2) for argument types \( \text{ArgTypes...} \) and return type \( R \).

8

Preconditions: \( F \) meets the \( \text{Cpp17CopyConstructible} \) requirements.

9

Postconditions: \( \!\!*this \) if any of the following hold:

9.1 — \( f \) is a null function pointer value.
9.2 — \( f \) is a null member pointer value.
9.3 — \( F \) is an instance of the \( \text{function} \) class template, and \( \!f \).

Otherwise, \( \!*this \) targets a copy of \( f \) initialized with \( \text{std::move}(f) \). [Note: 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. —end note]

Throws: Nothing if \( f \) is a specialization of \( \text{reference_wrapper} \) or a function pointer. Otherwise, may throw \( \text{bad_alloc} \) or any exception thrown by \( F \)'s copy or move constructor.

```
template<class F> function(F) -> function<
```

see below;

12

Constraints: \&\&\&::operator() is well-formed when treated as an unevaluated operand and \( \text{decltype(}\&\&\&::operator()) is of the form R(G::*)(A...) cv \&opt \) noexcept\_opt for a class type \( G \).

13

Remarks: The deduced type is \( \text{function}<R(A...)> \).

14

[Example:

dvoid f() {
  int i{5};
  function g = [\&](double) { return i; }; // deduces function<int(double)>
}

—end example]

```
function& operator=(const function& f);
```

15

Effects: As if by \( \text{function}(f).\text{swap}(\!*this) \);

16

Returns: \( \!*this \).

```
function& operator=(function&& f);
```

17

Effects: Replaces the target of \( \!*this \) with the target of \( f \).

18

Returns: \( \!*this \).

```
function& operator=(nullptr_t) noexcept;
```

19

Effects: If \( \!*this \neq \text{nullptr} \), destroys the target of this.

20

Postconditions: \( \!\!(\!*this) \).

21

Returns: \( \!*this \).

```
template<class F> function& operator=(F&& f);
```

22

Constraints: \( \text{decay_t}<F> \) is Lvalue-Callable (20.14.16.2) for argument types \( \text{ArgTypes...} \) and return type \( R \).

23

Effects: As if by: \( \text{function}(\text{std::forward}<F>(f)).\text{swap}(\!*this) \);

24

Returns: \( \!*this \).
template<class F> function& operator=(reference_wrapper<F> f) noexcept;

Effects: As if by: function(f).swap(*this);

Returns: *this.

~function();

Effects: If *this != nullptr, destroys the target of this.

### 20.14.16.2.2 Modifiers

void swap(function& other) noexcept;

Effect: Interchanges the targets of *this and other.

### 20.14.16.2.3 Capacity

explicit operator bool() const noexcept;

Returns: true if *this has a target, otherwise false.

### 20.14.16.2.4 Invocation

R operator()(ArgTypes... args) const;

Returns: INVOKER<R>(f, std::forward<ArgTypes>(args)...) (20.14.3), where f is the target object (20.14.2) of *this.

Throws: bad_function_call if ![this]; otherwise, any exception thrown by the wrapped callable object.

### 20.14.16.2.5 Target access

const type_info& target_type() const noexcept;

Returns: If *this has a target of type T, typeid(T); otherwise, typeid(void).

template<class T> T* target() noexcept;  
template<class T> const T* target() const noexcept;

Returns: If target_type() == typeid(T) a pointer to the stored function target; otherwise a null pointer.

### 20.14.16.2.6 Null pointer comparison functions

template<class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;

Returns: ![f].

### 20.14.16.2.7 Specialized algorithms

template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2) noexcept;

Effects: As if by: f1.swap(f2);

### 20.14.17 Searchers

This subclause 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.

Each specialization of a class template specified in this subclause 20.14.17 shall meet the Cpp17CopyConstructible and Cpp17CopyAssignable requirements. Template parameters named

(2.1) — ForwardIterator,
(2.2) — ForwardIterator1,
(2.3) — ForwardIterator2,
(2.4) — RandomAccessIterator,
— RandomAccessIterator1,
— RandomAccessIterator2, and
— BinaryPredicate
of templates specified in this subclause 20.14.17 shall meet the same requirements and semantics as specified in 25.1. Template parameters named Hash shall meet the Cpp17Hash requirements (Table 34).

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.

### 20.14.17.1 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`.

3 Effects: Returns a pair of iterators `i` and `j` such that

   — `i == search(first, last, pat_first_, pat_last_, pred_)`, and

   — if `i == last`, then `j == last`, otherwise `j == next(i, distance(pat_first_, pat_last_))`.

### 20.14.17.2 Class template boyer_moore_searcher

```cpp
template<class RandomAccessIterator1, 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
};
```

§ 20.14.17.2 686
§ 20.14.17.3 687

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.

Preconditions: 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 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.

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 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.
template<class RandomAccessIterator1, class RandomAccessIterator2>
operator()(RandomAccessIterator1 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

1. $i$ is the first iterator $i$ in the range $[\text{first, last} - (\text{pat_last_} - \text{pat_first}_))$ such that for every non-negative integer $n$ less than $\text{pat_last_} - \text{pat_first}_$ the following condition holds:

   
   $\text{pred}(*i + n, *(\text{pat_first_} + n)) != \text{false}$, and

2. $j == \text{next}(i, \text{distance}(\text{pat_first_}, \text{pat_last}_)).$

Returns make_pair(first, first) if $[\text{pat_first_}, \text{pat_last}_)$ is empty, otherwise returns make_pair(last, last) if no such iterator is found.

Complexity: At most $(\text{last} - \text{first}) \times (\text{pat_last_} - \text{pat_first}_)$ applications of the predicate.

## 20.14.18 Class template hash

The unordered associative containers defined in 22.5 use specializations of the class template hash (20.14.1) as the default hash function.

Each specialization of hash is either enabled or disabled, as described below. [Note: 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>. Disallowed specializations of hash are not function object types (20.14). [Note: 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:

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),

2. be swappable (16.5.3.2) for lvalues,

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;

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

This subclause 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.

2 All functions specified in this subclause are signal-safe (17.13.4).

20.15.1 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 (20.15.3), 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 (20.15.3), 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.

4 Unless otherwise specified, the behavior of a program that adds specializations for any of the templates specified in this subclause 20.15 is undefined.

5 Unless otherwise specified, an incomplete type may be used to instantiate a template specified in this subclause. The behavior of a program is undefined if:

— an instantiation of a template specified in subclause 20.15 directly or indirectly depends on an incompletely-defined object type T, and

— that instantiation could yield a different result were T hypothetically completed.

20.15.2 Header <type_traits> synopsis

namespace std {

// 20.15.3, 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>;

// 20.15.4.1, 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;

// 20.15.4.2, 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;

// 20.15.4.3, type properties
template<class T> struct is_const;
template<class T> struct is_volatiele;
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;
// 20.15.5, type property queries
template<class T> struct alignment_of;
template<class T> struct rank;
template<class T, unsigned I = 0> struct extent;

// 20.15.6, 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;

// 20.15.7.1, 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;

using remove_const_t = typename remove_const<T>::type;
using remove_volatile_t = typename remove_volatile<T>::type;
using remove_cv_t = typename remove_cv<T>::type;
using add_const_t = typename add_const<T>::type;
using add_volatile_t = typename add_volatile<T>::type;
using add_cv_t = typename add_cv<T>::type;

// 20.15.7.2, reference modifications
template<class T> struct remove_reference;
template<class T> struct add_lvalue_reference;
template<class T> struct add_rvalue_reference;

using remove_reference_t = typename remove_reference<T>::type;
using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// 20.15.7.3, 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;

// 20.15.7.4, 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;

// 20.15.7.5, 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;

// 20.15.7.6, other transformations
template<class T> struct type_identity;
template<size_t Len, size_t Align = default_alignment>  // see 20.15.7.6
  struct aligned_storage;
template<size_t Len, class... Types> struct aligned_union;
template<class T> struct remove_cvref;
template<class T> struct remove_cvref_t = typename remove_cvref<T>::type;
template<class T> struct decay;
template<class T> struct 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, 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>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T>
  using underlying_type_t = typename underlying_type<T>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;

template<class T>
  using type_identity_t = typename type_identity<T>::type;
template<size_t Len, size_t Align = default_alignment>  // see 20.15.7.6
  struct aligned_storage;
template<size_t Len, class... Types>
  struct aligned_union;
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>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T>
  using underlying_type_t = typename underlying_type<T>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;

template<class T>
  using type_identity_t = typename type_identity<T>::type;
template<size_t Len, size_t Align = default_alignment>  // see 20.15.7.6
  struct aligned_storage;
template<size_t Len, class... Types>
  struct aligned_union;
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>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T>
  using underlying_type_t = typename underlying_type<T>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;

template<class T>
  using type_identity_t = typename type_identity<T>::type;
template<size_t Len, size_t Align = default_alignment>  // see 20.15.7.6
  struct aligned_storage;
template<size_t Len, class... Types>
  struct aligned_union;
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>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T>
  using underlying_type_t = typename underlying_type<T>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_reference_t = typename common_reference<T, U, TQual, UQual>::type;
template<class T, class U, template<class> class TQual, template<class> class UQual>
  using common_type_t = typename common_type<T, U, TQual, UQual>::type;

§ 20.15.2
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...
  using void_t = void;

// 20.15.8, logical operator traits
template<class... B> struct conjunction;
template<class... B> struct disjunction;
template<class B> struct negation;

// 20.15.4.1, 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;

// 20.15.4.2, 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_member_pointer_v = is_member_pointer<T>::value;

// 20.15.4.3, 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_abstract_v = is_abstract<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, 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, class U>
inline constexpr bool is_swappable_v = is_swappable<T, U>::value;

-template<class T, class... Args>
inline constexpr bool is_swappable_with_v = is_swappable_with<T, Args...>::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>
    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 has_virtualDestructor_v = has_virtualDestructor<T>::value;

template<class T, class U>
    inline constexpr bool is_same_v = is_same<T, U>::value;

template<class Base, class Derived>
    inline constexpr bool is_base_of_v = is_base_of<Base, Derived>::value;

template<class From, class To>
    inline constexpr bool is_convertible_v = is_convertible<From, To>::value;

template<class From, class To>
    inline constexpr bool is_nothrow_convertible_v = is_nothrow_convertible<From, To>::value;

template<class Base, class Derived>
    inline constexpr bool is_layout_compatible_v = is_layout_compatible<T, U>::value;

template<class Base, class Derived>
    inline constexpr bool is_pointer_interconvertible_base_of_v = is_pointer_interconvertible_base_of<Base, Derived>::value;

template<class Fn, class... ArgTypes>
    inline constexpr bool is_invocable_v = is_invocable<Fn, ArgTypes...>::value;

template<class R, class Fn, class... ArgTypes>
    inline constexpr bool is_invocable_r_v = is_invocable_r<R, Fn, ArgTypes...>::value;

template<class R, class Fn, class... ArgTypes>
    inline constexpr bool is_nothrow_invocable_v = is_nothrow_invocable<R, Fn, ArgTypes...>::value;

template<class R, class Fn, class... ArgTypes>
    inline constexpr bool is_nothrow_invocable_r_v = is_nothrow_invocable_r<R, Fn, ArgTypes...>::value;
// 20.15.3 Helper classes [meta.help]
namespace std {
    template<class T, T v> struct integral_constant {
        static constexpr T value = v;

        using value_type = T;
        using type = integral_constant<T, v>;

        constexpr operator value_type() const noexcept { return value; }
        constexpr value_type operator()() const noexcept { return value; }
    };
}

// 20.15.4 Unary type traits [meta.unary]
This subclause contains templates that may be used to query the properties of a type at compile time.

Each of these templates shall be a Cpp17UnaryTypeTrait (20.15.1) with a base characteristic of true_type if
the corresponding condition is true, otherwise false_type.

20.15.4.1 Primary type categories [meta.unary.cat]
The primary type categories correspond to the descriptions given in subclause 6.8 of the C++ standard.

For any given type T, the result of applying one of these templates to T and to cv T shall yield the same
result.

[Note: 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 [tab:meta.unary.cat]

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>struct void;</td>
<td>T is void</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>struct is_void;</td>
<td>T is void</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>struct is_null_pointer;</td>
<td>T is nullptr_t (6.8.1)</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>struct is_integral;</td>
<td>T is an integral type (6.8.1)</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>struct is_floating_point;</td>
<td>T is a floating-point type (6.8.1)</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>struct is_array;</td>
<td>T is an array type (6.8.2) of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>known or unknown extent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Class template array (22.3.7) is not an</td>
</tr>
<tr>
<td></td>
<td></td>
<td>array type.</td>
</tr>
</tbody>
</table>
Table 47: Primary 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_pointer;</td>
<td>T is a pointer type (6.8.2)</td>
<td>Includes pointers to functions but not pointers to non-static members.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_lvalue_reference;</td>
<td>T is an lvalue reference type (9.3.3.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_rvalue_reference;</td>
<td>T is an rvalue reference type (9.3.3.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_object_pointer;</td>
<td>T is a pointer to data member</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_function_pointer;</td>
<td>T is a pointer to member function</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_enum;</td>
<td>T is an enumeration type (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_union;</td>
<td>T is a union type (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_class;</td>
<td>T is a non-union class type (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_function;</td>
<td>T is a function type (6.8.2)</td>
<td></td>
</tr>
</tbody>
</table>

20.15.4.2 Composite type traits

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>T is an lvalue reference or an rvalue reference</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_arithmetic;</td>
<td>T is an arithmetic type (6.8.1)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_fundamental;</td>
<td>T is a fundamental type (6.8.1)</td>
<td></td>
</tr>
<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 (6.8.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_pointer;</td>
<td>T is a pointer-to-member</td>
<td></td>
</tr>
</tbody>
</table>

20.15.4.3 Type properties

These templates provide access to some of the more important properties of types.

It is unspecified whether the library defines any full or partial specializations of any of these templates.

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.

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, 11.4.3) 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>
<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 (6.8.3)</td>
<td>remove all extents -t&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_volatile;</td>
<td>T is volatile-qualified (6.8.3)</td>
<td>remove all extents -t&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivial;</td>
<td>T is a trivial type (6.8)</td>
<td>remove all extents -t&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_copyable;</td>
<td>T is a trivially copyable type (6.8)</td>
<td>remove all extents -t&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_standard_layout;</td>
<td>T is a standard-layout type (6.8)</td>
<td>remove all extents -t&lt;T&gt; shall be a complete type or cv void.</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.</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 is_polymorphic;</td>
<td>T is a polymorphic class (11.7.2)</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 is_abstract;</td>
<td>T is an abstract class (11.7.3)</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 is_final;</td>
<td>T is a class type marked with the class-virt-specifier final (11.1). [Note: A union is a class type that can be marked with final. —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 (9.4.1)</td>
<td>remove all extents -t&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_signed;</td>
<td>If is_arithmetic_v&lt;T&gt; 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 is_arithmetic_v&lt;T&gt; 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 (9.3.3.4)</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 (9.3.3.4)</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 cv void type T, is_constructible_v&lt;T, Args...&gt; is false, otherwise see below</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</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_default_constructible;</td>
<td>is_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>
<tr>
<td>template&lt;class T&gt; struct is_copy_constructible;</td>
<td>For a referenceable type T (16.3.23), the same result as is_constructible_v&lt;T, 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_move_constructible;</td>
<td>For a referenceable type T, the same result as is_constructible_v&lt;T, 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_assignable;</td>
<td>The expression declval&lt;T&gt;() = declval&lt;U&gt;() 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: 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]</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_copy_assignable;</td>
<td>For a referenceable type T, the same result as is_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_move_assignable;</td>
<td>For a referenceable type T, the same result as is_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</td>
<td>Condition</td>
<td>Preconditions</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_swappable_with;</td>
<td>The expressions</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td></td>
<td>swap(declval&lt;T&gt;(), declval&lt;U&gt;()) and swap(declval&lt;U&gt;(), declval&lt;T&gt;()) are each well-formed when treated as an unevaluated operand (7.2) in an overload-resolution context for swappable values (16.5.3.2). Access checking is performed as if in a context unrelated to T and U. Only the validity of the immediate context of the swap expressions is considered. [Note: 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></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_swappable;</td>
<td>For a referenceable type T, the same result as is_swappable_with_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_destructible;</td>
<td>Either T is a reference type, or T is a complete object type for which the expression declval&lt;U&amp;&gt;() . ~U() is well-formed when treated as an unevaluated operand (7.2), where U is remove_all_extents_t&lt;T&gt;.</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>is_constructible_v&lt;T, Args...&gt; is true and the variable definition for is_constructible, as defined below, is known to call no operation that is not trivial (6.8, 11.4.3).</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</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_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>
<tr>
<td>template&lt;class T&gt; struct is_trivially_copy_constructible;</td>
<td>For a referenceable type T, the same result as is_trivially_constructible_v&lt;T, 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_trivially_move_constructible;</td>
<td>For a referenceable type T, the same result as is_trivially_constructible_v&lt;T, 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_trivially_assignable;</td>
<td>is_assignable_v&lt;T, U&gt; is true and the assignment, as defined by is_assignable, is known to call no operation that is not trivial (6.8, 11.4.3).</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_trivially_copy_assignable;</td>
<td>For a referenceable type T, the same result as is_trivially_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_trivially_move_assignable;</td>
<td>For a referenceable type T, the same result as is_trivially_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&gt; struct is_trivially_destructible;</td>
<td>is_destructible_v&lt;T&gt; is true and 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, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class... Args&gt; struct is_nothrow_constructible;</td>
<td>is_constructible_v&lt;T, Args...&gt; is true and the variable definition for is_constructible, as defined below, is known not to throw any exceptions (7.6.2.6).</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_nothrow_default_constructible;</td>
<td>is_nothrow_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>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_copy_constructible;</td>
<td>For a referenceable type T, the same result as is_nothrow_constructible_v&lt;T, const T&amp;&gt;, otherwise false.</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><code>template&lt;class T&gt; struct is_nothrow_move_constructible;</code></td>
<td>For a referenceable type T, the same result as <code>is_nothrow_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>
<tr>
<td><code>template&lt;class T, class U&gt; struct is_nothrow_assignable;</code></td>
<td><code>is_assignable_v&lt;T, U&gt;</code> is true and the assignment is known not to throw any exceptions (7.6.2.6).</td>
<td>T and U shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_nothrow_copy_assignable;</code></td>
<td>For a referenceable type T, the same result as <code>is_nothrow_assignable_v&lt;T&amp;, 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><code>template&lt;class T&gt; struct is_nothrow_move_assignable;</code></td>
<td>For a referenceable type T, the same result as <code>is_nothrow_assignable_v&lt;T&amp;, 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>
<tr>
<td><code>template&lt;class T, class U&gt; struct is_nothrow_swappable_with;</code></td>
<td><code>is_swappable_with_v&lt;T, U&gt;</code> is true and each swap expression of the definition of <code>is_swappable_with_v&lt;T, U&gt;</code> is known not to throw any exceptions (7.6.2.6).</td>
<td>T and U shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_nothrow_swappable;</code></td>
<td>For a referenceable type T, the same result as <code>is_nothrow_swappable_v&lt;T&amp;, 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><code>template&lt;class T&gt; struct is_nothrow_destructible;</code></td>
<td><code>is_destructible_v&lt;T&gt;</code> is true and the indicated destructor is known not to throw any exceptions (7.6.2.6).</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct has_virtual_destructor;</code></td>
<td>T has a virtual destructor (11.4.6)</td>
<td>If T is a non-union class type, T shall be a complete type.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct has_unique_object_representations;</code></td>
<td>For an array type T, the same result as <code>has_unique_object_representations_v&lt;remove_all_extents_v&lt;T[]&gt;, otherwise see below</code></td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
</tbody>
</table>

5 [Example:

```cpp
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:
Example:

```cpp
// 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

8 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`:

```cpp
T t(declval<Args>()...);
```

[Note: 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: 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]

9 The predicate condition for a template specialization `has_unique_object_representations<T>` shall be satisfied if and only if:

(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: If a type has padding bits, the condition does not hold; otherwise, the condition holds true for integral types. —end note]

20.15.5 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><code>template&lt;class T&gt;</code></td>
<td><code>alignof(T)</code></td>
</tr>
<tr>
<td><code>struct alignment_of;</code></td>
<td><code>Mandates: alignof(T) is a valid expression (7.6.2.5)</code></td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td><code>If T names an array type, an integer value representing the number of dimensions of T; otherwise, 0.</code></td>
</tr>
<tr>
<td><code>struct rank;</code></td>
<td><code>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 (9.3.3.4) of the I_th dimension of T, where indexing of I is zero-based</code></td>
</tr>
</tbody>
</table>

2 Each of these templates shall be a `Cpp17UnaryTypeTrait (20.15.1)` with a base characteristic of `integral_constant<size_t, Value>`.

3 [Example:
// the following assertions hold:
assert(rank_v<int> == 0);
assert(rank_v<int[2]> == 1);
assert(rank_v<int[int[4]> == 2);
— end example]

[Example:
// 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[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[int[4], 1>) == 4);
— end example]

20.15.6 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` (20.15.1) 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>template&lt;class T, class U&gt; struct is_same;</td>
<td>T and U name the same type with the same cv-qualifications</td>
<td></td>
</tr>
<tr>
<td>template&lt;class Base, class Derived&gt; struct is_base_of;</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. [Note: Base classes that are private, protected, or ambiguous are, nonetheless, base classes. — end note]</td>
</tr>
<tr>
<td>template&lt;class From, class To&gt; struct is_convertible;</td>
<td>see below</td>
<td>From and To shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class From, class To&gt; struct is_nothrow_convertible;</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 (7.6.2.6)</td>
<td>From and To shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_layout_compatible;</td>
<td>T and U are layout-compatible (6.8)</td>
<td>T and U shall be complete types, <code>cv void</code>, or arrays of unknown bound.</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><code>template&lt;class Base, class Derived&gt;</code></td>
<td><code>is_pointer_interconvertible_base_of;</code></td>
<td><code>Derived is unambiguously derived from Base without regard to cv-qualifiers, and each object of type Derived is pointer-interconvertible (6.8.2) with its Base subobject, or Base and Derived are not unions and name the same class type without regard to cv-qualifiers. 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.</code></td>
</tr>
<tr>
<td><code>template&lt;class Fn, class... ArgTypes&gt;</code></td>
<td><code>is_invocable;</code></td>
<td><code>The expression </code>INVOKE(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...)<code>is well-formed when treated as an unevaluated operand. Fn and all types in the template parameter pack</code>ArgTypes<code>shall be complete types, cv</code>void<code>, or arrays of unknown bound.</code></td>
</tr>
<tr>
<td><code>template&lt;class R, class Fn, class... ArgTypes&gt;</code></td>
<td><code>is_invocable_r;</code></td>
<td><code>The expression </code>INVOKE&lt;R&gt;(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...)<code>is well-formed when treated as an unevaluated operand. Fn, R, and all types in the template parameter pack</code>ArgTypes<code>shall be complete types, cv</code>void<code>, or arrays of unknown bound.</code></td>
</tr>
<tr>
<td><code>template&lt;class Fn, class... ArgTypes&gt;</code></td>
<td><code>is_nothrow_invocable;</code></td>
<td><code>is_invocable_v&lt;Fn, ArgTypes...&gt; is true and the expression </code>INVOKE(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is known not to throw any exceptions (7.6.2.6). Fn and all types in the template parameter pack <code>ArgTypes</code> shall be complete types, cv <code>void</code>, or arrays of unknown bound.`</td>
</tr>
<tr>
<td><code>template&lt;class R, class Fn, class... ArgTypes&gt;</code></td>
<td><code>is_nothrow_invocable_r;</code></td>
<td><code>is_invocable_r_v&lt;R, Fn, ArgTypes...&gt; is true and the expression </code>INVOKE&lt;R&gt;(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is known not to throw any exceptions (7.6.2.6). Fn, R, and all types in the template parameter pack <code>ArgTypes</code> shall be complete types, cv <code>void</code>, or arrays of unknown bound.`</td>
</tr>
</tbody>
</table>

3 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, 11.4.3) 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.

4 Example:
```cpp
struct B {};  // true
struct B1 : B {}; // true
struct B2 : B {}; // true
struct D : private B1, private B2 {}; // false
```

— end example
The predicate condition for a template specialization \texttt{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: 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 \texttt{To} and \texttt{From}. Only the validity of the immediate context of the \texttt{return} statement (8.7.3) (including initialization of the returned object or reference) is considered. [Note: 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]

20.15.7 Transformations between types \[meta.trans\]

This subclause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this subclause shall be a \texttt{Cpp17TransformationTrait} (20.15.1).

### 20.15.7.1 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><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>struct remove_const;</td>
<td>The member typedef \texttt{type} names the same type as \texttt{T} except that any top-level cv-qualifier has been removed. [Example: <code>remove\_const\_t&lt;const volatile int&gt;</code> evaluates to <code>volatile int</code>, whereas <code>remove\_const\_t&lt;const int*&gt;</code> evaluates to <code>const int*</code>. — end example]</td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>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. [Example: <code>remove\_volatile\_t&lt;const volatile int&gt;</code> evaluates to <code>const int</code>, whereas <code>remove\_volatile\_t&lt;volatile int*&gt;</code> evaluates to <code>volatile int*</code>. — end example]</td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>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. [Example: <code>remove\_cv\_t&lt;const volatile int&gt;</code> evaluates to <code>int</code>, whereas <code>remove\_cv\_t&lt;volatile int*&gt;</code> evaluates to <code>volatile int*</code>. — end example]</td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>struct add_const;</td>
<td>If \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><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>struct add_volatile;</td>
<td>If \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><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>struct add_cv;</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

20.15.7.2 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><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>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><code>template&lt;class T&gt;</code></td>
<td></td>
</tr>
<tr>
<td>struct add_lvalue_reference;</td>
<td>If \texttt{T} names a referenceable type (16.3.23) then the member typedef \texttt{type} names \texttt{T}; otherwise, \texttt{type} names \texttt{T}. [Note: This rule reflects the semantics of reference collapsing (9.3.3.2). — end note]</td>
</tr>
</tbody>
</table>

§ 20.15.7.2
Table 53: Reference modifications (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>If T names a referenceable type then the member typedef type names T&amp;&amp;; otherwise, type names T. [Note: This rule reflects the semantics of reference collapsing (9.3.3.2). For example, when a type T names a type T[], the type add_rvalue_reference_t.&lt;T&gt; is not an rvalue reference. — end note]</td>
</tr>
</tbody>
</table>

20.15.7.3 Sign modifications

Table 54: Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>If T names a (possibly cv-qualified) signed integer type (6.8.1) then the member typedef type names the type T; otherwise, if T names a (possibly cv-qualified) unsigned integer type then type names the corresponding signed integer type, with the same cv-qualifiers as T; otherwise, type names the signed integer type with smallest rank (6.8.4) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. <strong>Mandates:</strong> T is an integral or enumeration type other than cv bool.</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>If T names a (possibly cv-qualified) unsigned integer type (6.8.1) then the member typedef type names the type T; otherwise, if T names a (possibly cv-qualified) signed integer type then type names the corresponding unsigned integer type, with the same cv-qualifiers as T; otherwise, type names the unsigned integer type with smallest rank (6.8.4) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. <strong>Mandates:</strong> T is an integral or enumeration type other than cv bool.</td>
</tr>
</tbody>
</table>

20.15.7.4 Array modifications

Table 55: Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>If T names a type “array of U”, the member typedef type shall be U, otherwise T. [Note: 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]</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>If T is “multi-dimensional array of U”, the resulting member typedef type is U, otherwise T.</td>
</tr>
</tbody>
</table>

1 [Example:

```
// the following assertions hold:
assert((is_same_v<remove_extent<int>, int>));
assert((is_same_v<remove_extent<int[2]>, int>));
assert((is_same_v<remove_extent<int[2][3]>, int[3]>));
assert((is_same_v<remove_extent<int[1][3]>, int[3]>));
— end example]
```

2 [Example:

```
// the following assertions hold:
assert((is_same_v<remove_all_extents<int>, int>));
assert((is_same_v<remove_all_extents<int[2]>, int>));
assert((is_same_v<remove_all_extents<int[2][3]>, int>));
assert((is_same_v<remove_all_extents<int[1][3]>, int>));
— end example]
```
## 20.15.7.5 Pointer modifications

### Table 56: Pointer modifications

<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 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 (16.3.23) or a cv void type then the member typedef type names the same type as remove_reference_t&lt;T&gt;*; otherwise, type names T.</td>
</tr>
</tbody>
</table>

## 20.15.7.6 Other transformations

### Table 57: Other transformations

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct type_identity;</td>
<td>The member typedef type names the type T.</td>
</tr>
</tbody>
</table>
| template<
  size_t Len,
  size_t Align = default-alignment
> struct aligned_storage; | The value of default-alignment shall be the most stringent alignment requirement for any C++ object type whose size is no greater than Len (6.8). The member typedef type shall be a trivial standard-layout type suitable for use as uninitialized storage for any object whose size is at most Len and whose alignment is a divisor of Align. Mandates: Len is not zero. Align is equal to alignof(T) for some type T or to default-alignment. |
| template<
  size_t Len,
  class... Types>
struct aligned_union; | The member typedef type shall be a trivial standard-layout type suitable for use as uninitialized storage for any object whose type is listed in Types; its size shall be at least Len. The static member alignment_value shall be an integral constant of type size_t whose value is the strictest alignment of all types listed in Types. Mandates: At least one type is provided. Each type in the template parameter pack Types is a complete object type. |
| template<class T> struct remove_cvref; | The member typedef type names the same type as remove_cv_t<remove_reference_t<T>>. |
| template<class T> struct decay; | Let U be remove_reference_t<T>. If is_array_v<U> is true, the member typedef type equals remove_extent_t<U>*. If is_function_v<U> is true, the member typedef type equals add_pointer_t<U>. Otherwise the member typedef type equals remove_cv_t<U>. [Note: This behavior is similar to the value-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) conversions applied when an value 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] |
| template<bool B, class T = void> struct enable_if; | If B is true, the member typedef type shall equal T; otherwise, there shall be no member type. |
| template<bool B, class T, class F> struct conditional; | If B is true, the member typedef type shall equal T. If B is false, the member typedef type shall equal F. |
| template<class... T> struct common_type; | Unless this trait is specialized (as specified in Note B, below), the member type is defined or omitted as specified in Note A, below. If it is omitted, there shall be no member type. Each type in the template parameter pack T shall be complete, cv void, or an array of unknown bound. |
| template<class, class, template<class> class, template<class> class>
struct basic_common_reference; | Unless this trait is specialized (as specified in Note D, below), there shall be no member type. |
Table 57: Other transformations (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class... T&gt; struct common_reference;</td>
<td>The member typedef-name type is defined or omitted as specified in Note C, below. Each type in the parameter pack T shall be complete or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct underlying_type;</td>
<td>If T is an enumeration type, the member typedef type names the underlying type of T (9.7.1); otherwise, there is no member type. Mandates: T is not an incomplete enumeration type.</td>
</tr>
<tr>
<td>template&lt;class Fn, class... ArgTypes&gt; struct invoke_result;</td>
<td>If the expression INVOKE(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is well-formed when treated as an unevaluated operand (7.2), the member typedef type names the type decltype(INVOKE(declval&lt;Fn&gt;(), 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. [Note: 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]</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct unwrap_reference;</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>template&lt;class T&gt; unwrap_ref_decay;</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>
</tbody>
</table>

1 [Note: 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> (20.14.1) is included.

3 Let:

(3.1) — CREFA be add_lvalue_reference_t<const remove_reference_t<A>>,

(3.2) — XREF(A) denote a unary alias template T such that TC<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,

(3.3) — COPYCV(FROM, TO) be an alias for type TO with the addition of FROM’s top-level cv-qualifiers, [Example: COPYCV(const int, volatile short) is an alias for const volatile short. — end example]

(3.4) — 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:

(3.5) — If A and B are both lvalue reference types, COMMON-REF(A, B) is COND-RES(COPYCV(X, Y) &copycv(Y, X) &) if that type exists and is a reference type.

(3.6) — Otherwise, let C be remove_reference_t<COMMON-REF(Xk, Yk)k&k. If A and B are both rvalue reference types, C is well-formed, and is_convertible_v<A, C> && is_convertible_v<B, C> is true, then COMMON-REF(A, B) is C. 

§ 20.15.7.6
— Otherwise, let D be $\text{COMMON-REF}(\text{const } X_k, Y_k)$. If A is an rvalue reference and B is a value reference and D is well-formed and is_convertible_v<A, D> is true, then $\text{COMMON-REF}(A, B)$ is D.

— Otherwise, if A is a value reference and B is an rvalue reference, then $\text{COMMON-REF}(A, B)$ is $\text{COMMON-REF}(B, A)$.

— Otherwise, $\text{COMMON-REF}(A, B)$ is ill-formed.

If any of the types computed above is ill-formed, then $\text{COMMON-REF}(A, B)$ is ill-formed.

4 Note A: For the $\text{common_type}$ trait applied to a template parameter pack T of types, the member $\text{type}$ shall be either defined or not present as follows:

(4.1) — If sizeof...(T) is zero, there shall be no member $\text{type}$.

(4.2) — If sizeof...(T) is one, let T0 denote the sole type constituting the pack T. The member typedef-name $\text{type}$ shall denote the same type, if any, as $\text{common_type}_{-}\text{t<T0, T0>}$. otherwise there shall be no member $\text{type}$.

(4.3) — If sizeof...(T) is two, let the first and second types constituting T be denoted by T1 and T2, respectively, and let D1 and D2 denote the same types as decay_t<T1> and decay_t<T2>, respectively.

(4.3.1) — If is_same_v<T1, D1> is false or is_same_v<T2, D2> is false, let C denote the same type, if any, as common_type_t<D1, D2>.

(4.3.2) — [Note: None of the following will apply if there is a specialization $\text{common_type}<D1, D2>$. — end note]

(4.3.3) — Otherwise, if

\[
\text{decay_t<decltype(false ? declval<D1>() : declval<D2>())>
\]

denotes a valid type, let C denote that type.

(4.3.4) — Otherwise, if $\text{COND-RES}(\text{CREF}(D1), \text{CREF}(D2))$ denotes a type, let C denote the type decay_t<$\text{COND-RES}(\text{CREF}(D1), \text{CREF}(D2))$>.

In either case, the member typedef-name $\text{type}$ shall denote the same type, if any, as C. Otherwise, there shall be no member $\text{type}$.

(4.4) — If sizeof...(T) is greater than two, let T1, T2, and R, respectively, denote the first, second, and (pack of) remaining types constituting T. Let C denote the same type, if any, as $\text{common_type}_{-}\text{t<T1, T2>}$ and T0 denote the sole type in the pack T. The member typedef-name $\text{type}$ shall denote the same type, if any, as $\text{common_type}_{-}\text{t<C, R,...>}$. Otherwise, there shall be no member $\text{type}$.

5 Note B: Notwithstanding the provisions of 20.15.2, and pursuant to 16.5.4.2.1, a program may specialize $\text{common_type}_{<T1, T2>}$ for types T1 and T2 such that is_same_v<T1, decay_t<T1>> and is_same_v<T2, decay_t<T2>> are each true. [Note: 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 $\text{type}$, 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 T1 and T2 is explicitly convertible. Moreover, $\text{common_type}_{-}\text{t<T1, T2>}$ shall denote the same type, if any, as does $\text{common_type}_{-}\text{t<T2, T1>}$.

No diagnostic is required for a violation of this Note’s rules.

6 Note C: For the $\text{common_reference}$ trait applied to a parameter pack T of types, the member $\text{type}$ shall be either defined or not present as follows:

(6.1) — If sizeof...(T) is zero, there shall be no member $\text{type}$.

(6.2) — Otherwise, if sizeof...(T) is one, let T0 denote the sole type in the pack T. The member typedef-type $\text{type}$ shall denote the same type as T0.

(6.3) — Otherwise, if sizeof...(T) is two, let T1 and T2 denote the two types in the pack T. Then

(6.3.1) — If T1 and T2 are reference types and $\text{COMMON-REF}(T1, T2)$ is well-formed, then the member typedef-type $\text{type}$ denotes that type.

(6.3.2) — Otherwise, if $\text{basic_common_reference}_{\text{remove_cvref}_t<T1>}, \text{remove_cvref}_t<T2>, \text{XREF}(T1), \text{XREF}(T2)>$::type is well-formed, then the member typedef-type $\text{type}$ denotes that type.

(6.3.3) — Otherwise, if $\text{COND-RES}(T1, T2)$ is well-formed, then the member typedef-type $\text{type}$ denotes that type.

(6.3.4) — Otherwise, if $\text{common_type}_{-}\text{t<T1, T2>}$ is well-formed, then the member typedef-type $\text{type}$ denotes that type.
— Otherwise, there shall be no member type.

— Otherwise, if sizeof...(T) is greater than two, let T1, T2, and Rest, respectively, denote the first, second, and (pack of) remaining types comprising T. Let C be the type common_reference_t<T1, T2>. Then:

— 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 20.15.2, and pursuant to 16.5.4.2.1, 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: 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: 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<PMF, const S*, const char&>);
```

— end example]

20.15.8 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 B_i for which bool(B_i::value) is false, then instantiating conjunction<B1, ..., BN>::value does not require the instantiation of B_j::value for j > i. [Note: This is analogous to the short-circuiting behavior of the built-in operator &&. — end note]

Every template type argument for which B_i::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 B_i in the list true_type, B_1, ..., B_N for which bool(B_i::value) is false, or

— if there is no such B_i, the last type in the list.
The member names of the base class, other than conjunction and operator\&\&, shall not be hidden and shall be unambiguously available in conjunction.

```
template<class... B> struct disjunction : see below {};
```

The class template disjunction forms the logical disjunction of its template type arguments.

For a specialization disjunction\textlangle B\textsubscript{1}, ..., B\textsubscript{N}\textrangle, if there is a template type argument B\textsubscript{i} for which bool(B\textsubscript{i}::value) is true, then instantiating disjunction\textlangle B\textsubscript{1}, ..., B\textsubscript{N}\textrangle::value does not require the instantiation of B\textsubscript{j}::value for \( j > i \). [Note: This is analogous to the short-circuiting behavior of the built-in operator |]. —end note]

Every template type argument for which B\textsubscript{i}::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\textlangle B\textsubscript{1}, ..., B\textsubscript{N}\textrangle has a public and unambiguous base that is either

\begin{enumerate}
\item the first type \( B\textsubscript{i} \) in the list false\_type, \( B\textsubscript{1}, ..., B\textsubscript{N} \), for which bool(\( B\textsubscript{i} \)::value) is true, or
\item if there is no such \( B\textsubscript{i} \), the last type in the list.
\end{enumerate}

[Note: 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\textlangle B\textrangle is a Cpp17UnaryTypeTrait with a base characteristic of bool\_constant<!bool(B::value)>.

## 20.15.9 Member relationships

```
template<class S, class M>
constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;
```

Mandates: S is a complete type.

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 (6.8.2) 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;
```

Mandates: S1 and S2 are complete types.

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: 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:]

```
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 "pointer to member of B of type int."
static_assert( is_pointer_interconvertible_with_class\textlangle C\textrangle( &C::b ) ); // Forces the use of class C, and fails.
static_assert( is_corresponding_member( &C::a, &C::b ) );
```

§ 20.15.9 712
// Succeeds because, despite its appearance, &C::a and &C::b have types
// “pointer to member of A of type int” and
// “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.

—end example] — end note

20.15.10 Constant evaluation context [meta.const.eval]

constexpr bool is_constant_evaluated() noexcept;

1 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).

2 [Example:

constexpr void f(unsigned char *p, int n) {
    if (std::is_constant_evaluated()) {
        // should not be a constexpr 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 Compile-time rational arithmetic [ratio]

20.16.1 In general [ratio.general]

1 Subclause 20.16 describes the ratio library. It provides a class template ratio which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type intmax_t.

2 Throughout subclause 20.16, the names of template parameters are used to express type requirements. If a template parameter is named R1 or R2, and the template argument is not a specialization of the ratio template, the program is ill-formed.

20.16.2 Header <ratio> synopsis [ratio.syn]

namespace std {

    // 20.16.3, class template ratio
    template<intmax_t N, intmax_t D = 1> class ratio;

    // 20.16.4, ratio arithmetic
    template<class R1, class R2> using ratio_add = see below;
    template<class R1, class R2> using ratio_subtract = see below;
    template<class R1, class R2> using ratio_multiply = see below;
    template<class R1, class R2> using ratio_divide = see below;

    // 20.16.5, ratio comparison
    template<class R1, class R2> struct ratio_equal;
    template<class R1, class R2> struct ratio_not_equal;
    template<class R1, class R2> struct ratio_less;
    template<class R1, class R2> struct ratio_less_equal;
    template<class R1, class R2> struct ratio_greater;
    template<class R1, class R2> struct ratio_greater_equal;

    template<class R1, class R2>
    inline constexpr bool ratio_equal_v = ratio_equal<R1, R2>::value;
    template<class R1, class R2>
    inline constexpr bool ratio_not_equal_v = ratio_not_equal<R1, R2>::value;
    template<class R1, class R2>
    inline constexpr bool ratio_less_v = ratio_less<R1, R2>::value;
    template<class R1, class R2>
    inline constexpr bool ratio_less_equal_v = ratio_less_equal<R1, R2>::value;
    template<class R1, class R2>
    inline constexpr bool ratio_greater_v = ratio_greater<R1, R2>::value;

    § 20.16.2 713
template<class R1, class R2>
inline constexpr bool ratio_greater_equal_v = ratio_greater_equal<R1, R2>::value;

// 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'; // see below
using atto = ratio<1, 1'000'000'000'000'000'000';
using femto = ratio<1, 1'000'000'000';
using pico = ratio<1, 1'000';
using nano = ratio<1, 1000';
using micro = ratio<1, 1000';
using milli = ratio<1, 1000';
using centi = ratio<1, 100';
using deci = ratio<1, 10';
using deca = ratio<1, 10';
using hecto = ratio<1, 1000';
using kilo = ratio<1, 1000';
using mega = ratio<1, 1000000';
using giga = ratio<1, 1000000000';
using tera = ratio<1, 1000000000000';
using peta = ratio<1, 1000000000000000';
using exa = ratio<1, 1000000000000000000';
using zetta = ratio<1, 1000000000000000000000000'; // see below
using yotta = ratio<10000000000000000000000000000000000'; // see below
}

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: 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:

\[
\begin{align*}
(2.1) \quad \text{num shall have the value } & \text{sign}(N) \ast \text{sign}(D) \ast \text{abs}(N) / \text{gcd}. \\
(2.2) \quad \text{den shall have the value } & \text{abs}(D) / \text{gcd}.
\end{align*}
\]

20.16.4 Arithmetic on ratios

1 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.

3 [Example:

\[
\begin{align*}
\text{static assert(ratio_add(ratio<1, 3>, ratio<1, 6>::num == 1, "1/3+1/6 == 1/2");} \\
\text{static assert(ratio_add(ratio<1, 3>, ratio<1, 6>::den == 2, "1/3+1/6 == 1/2");} \\
\text{static assert(ratio_multiply(ratio<1, 3>, ratio<3, 2>::num == 1, "1/3*3/2 == 1/2");} \\
\text{static assert(ratio_multiply(ratio<1, 3>, ratio<3, 2>::den == 2, "1/3*3/2 == 1/2");}
\end{align*}
\]
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 \times R2::den + R1::den \times R2::den)</td>
<td>(R2::num \times R1::den)</td>
</tr>
<tr>
<td>ratio_subtract&lt;R1, R2&gt;</td>
<td>(R1::num \times R2::den - R1::den \times R2::den)</td>
<td>(R2::num \times R1::den)</td>
</tr>
<tr>
<td>ratio_multiply&lt;R1, R2&gt;</td>
<td>(R1::num \times R2::num)</td>
<td>(R1::den \times R2::den)</td>
</tr>
<tr>
<td>ratio_divide&lt;R1, R2&gt;</td>
<td>(R1::num \times R2::num)</td>
<td>(R1::den \times R2::num)</td>
</tr>
</tbody>
</table>

// 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");
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

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 \(R1::num \times R2::den\) is less than \(R2::num \times R1::den\), 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

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

#include <compare> // see 17.11.1

namespace std {
    class type_index;
    template<class T> struct hash;
    template<> struct hash<type_index>;
}
20.17.2 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).

20.17.3 type_index members

type_index(const type_info& rhs) noexcept;

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().

cast 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:

```cpp
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: Because different parallel architectures may require idiosyncratic parameters for efficient execution, implementations may provide additional execution policies to those described in this standard as extensions. — end note]

20.18.2 Header <execution> synopsis

```cpp
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

```cpp
template<class T> struct is_execution_policy { see below }; 

is_execution_policy can be used to detect execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.
```
is_execution_policy<T> is a Cpp17UnaryTypeTrait with a base characteristic of true_type if T is the type of a standard or implementation-defined execution policy, otherwise false_type.

[Note: This provision reserves the privilege of creating non-standard execution policies to the library implementation. — end note]

The behavior of a program that adds specializations for is_execution_policy is undefined.

20.18.4 Sequenced execution policy

class execution::sequenced_policy { unspecified }; 1

The class 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.

2 During the execution of a parallel algorithm with the execution::sequenced_policy policy, if the invocation of an element access function exits via an uncaught exception, terminate() is called.

20.18.5 Parallel execution policy

class execution::parallel_policy { unspecified }; 1

The class 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.

2 During the execution of a parallel algorithm with the execution::parallel_policy policy, if the invocation of an element access function exits via an uncaught exception, terminate() is called.

20.18.6 Parallel and unsequenced execution policy

class execution::parallel_unsequenced_policy { unspecified }; 1

The class 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.

2 During the execution of a parallel algorithm with the execution::parallel_unsequenced_policy policy, if the invocation of an element access function exits via an uncaught exception, terminate() is called.

20.18.7 Unsequenced execution policy

class execution::unsequenced_policy { unspecified }; 1

The class 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.

2 During the execution of a parallel algorithm with the execution::unsequenced_policy policy, if the invocation of an element access function exits via an uncaught exception, terminate() is called.

20.18.8 Execution policy objects

inline constexpr execution::sequenced_policy execution::seq{ unspecified }; 1
inline constexpr execution::parallel_policy execution::par{ unspecified }; 1
inline constexpr execution::parallel_unsequenced_policy execution::par_unseq{ unspecified }; 1
inline constexpr execution::unsequenced_policy execution::unseq{ unspecified }; 1

1 The header <execution> declares global objects associated with each type of execution policy.

20.19 Primitive numeric conversions

20.19.1 Header <charconv> synopsis

namespace std { 1
// floating-point format for primitive numerical conversion
enum class chars_format { 1
scientific = unspecified,
fixed = unspecified,
}; 1
} 1

§ 20.19.1
The type `chars_format` is a bitmask type (16.4.2.2.4) with elements `scientific`, `fixed`, and `hex`.

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

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 `ptr` is the one-past-the-end pointer of the characters written. Otherwise, the member `ec` has the value `errc::value_too_large`, the member `ptr` has the value `last`, and the contents of the range `[first, last)` are unspecified.

The functions that take a floating-point `value` but not a `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 `from_chars` function recovers `value` exactly. [Note: This guarantee applies only if `to_chars` and `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 round_to_nearest (17.3.4.1).

3 The functions taking a chars_format parameter determine the conversion specifier for printf as follows:
The conversion specifier is f if fmt is chars_format::fixed, e if fmt is chars_format::scientific, a
(without leading "0x" in the result) if fmt is chars_format::hex, and g if fmt is chars_format::general.

4 to_chars_result to_chars(char* first, char* last, see below value, int base = 10);

4 Preconditions: base has a value between 2 and 36 (inclusive).
5 Effects: The value of 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
value is less than zero, the representation starts with '-'.
6 Throws: Nothing.
7 Remarks: The implementation shall provide overloads for all signed and unsigned integer types and
char as the type of the parameter value.

8 to_chars_result to_chars(char* first, char* last, float value);
9 to_chars_result to_chars(char* first, char* last, double value);
10 to_chars_result to_chars(char* first, char* last, long double value);

8 Effects: value is converted to a string in the style of printf in the "C" locale. The conversion specifier
is f or e, chosen according to the requirement for a shortest representation (see above); a tie is resolved
in favor of f.
9 Throws: Nothing.

10 to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
11 to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
12 to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

10 Preconditions: fmt has the value of one of the enumerators of chars_format.
11 Effects: value is converted to a string in the style of printf in the "C" locale.
12 Throws: Nothing.

13 to_chars_result to_chars(char* first, char* last, float value,
chars_format fmt, int precision);
14 to_chars_result to_chars(char* first, char* last, double value,
chars_format fmt, int precision);
15 to_chars_result to_chars(char* first, char* last, long double value,
chars_format fmt, int precision);

13 Preconditions: fmt has the value of one of the enumerators of chars_format.
14 Effects: value is converted to a string in the style of printf in the "C" locale with the given precision.
15 Throws: Nothing.

See also: ISO C 7.21.6.1

20.19.3 Primitive numeric input conversion [charconv.from.chars]

All functions named from_chars analyze the string [first, last) for a pattern, where [first, last) is
required to be a valid range. If no characters match the pattern, value is unmodified, the member ptr of
the return value is first and the member ec is equal to errc::invalid_argument. [Note: 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.
from_chars_result from_chars(const char* first, const char* last,  
    see below & 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.

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: 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 [format]

20.20.1 Header <format> synopsis [format.syn]

namespace std {
    // 20.20.4, formatting functions
    template<class... Args>
    string format(string_view fmt, const Args&... args);
    template<class... Args>
    wstring format(wstring_view fmt, const Args&... args);
    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);

    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);

    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);

§ 20.20.1 721
template<class Out, class... Args>
  Out format_to(Out out, const locale& loc, wstring_view fmt, const Args&... args);

template<Out>
  Out vformat_to(Out out, string_view fmt,
                  format_args_t<type_identity_t<Out>, char> args);

template<Out>
  Out vformat_to(Out out, wstring_view fmt,
                  format_args_t<type_identity_t<Out>, wchar_t> args);

template<Out>
  Out vformat_to(Out out, const locale& loc, string_view fmt,
                  format_args_t<type_identity_t<Out>, char> args);

template<Out>
  Out vformat_to(Out out, const locale& loc, wstring_view fmt,
                  format_args_t<type_identity_t<Out>, wchar_t> args);

template<Out>
  struct format_to_n_result {
    Out out;
    iter_difference_t<Out> size;
  };

template<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<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<Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, const locale& loc, string_view fmt, const Args&... args);

template<Out, class... Args>
  format_to_n_result<Out> format_to_n(Out out, const locale& loc, wstring_view fmt, const Args&... args);

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);

// §20.20.5, formatter
template<class T, class charT = char> struct formatter;

// §20.20.3, 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);
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

#### 20.20.2.1 In general

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

format-specifier:

: format-spec

format-spec: as specified by the formatter specialization for the argument type

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.

[Example:

```cpp
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: A format string cannot contain a mixture of automatic and manual indexing. — end note] [Example:

```cpp
class std::format_print
{
public:
    template <class F, class B, class S> 
    std::basic_string < F > print ( F f, B b, S s = S () ) const
    { return std::basic_string < F > ( std::basic_format_string < F > ( f, b, s ) ) ; }
};

std::format_print * mp; 
std::basic_string < char > s = mp -> print ( std::basic_format_string < char > ( my_format ) , my_base ) ;
```
— 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:

(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 [format.string.std] 1 Each formatter specializations described in 20.20.5.2 for fundamental and string types interprets format-spec as a std-format-spec. [Note: 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:

```cpp
std-format-spec:
    fill-and-align, align
fill-and-align:
    fill, align
fill:
    any character other than { or }
align: one of < > ^
sign: one of + - space
width:
    positive-integer
        { arg-idopt }
precision:
    . nonnegative-integer
        { arg-idopt }
type: one of a A b B c d e E f F g G o p s x X
```

2 [Note: 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:

```cpp
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 "x*****"
string s3 = format("{:*>6}", 'x');  // value of s3 is "*****x"
string s4 = format("{:*^6}", 'x');  // value of s4 is "**x***"
string s5 = format("{:6d}", c);    // value of s5 is " 120"
string s6 = format("{:6}", true);  // value of s6 is "true ".

— end example] [Note: 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  [tab:format.align]

<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/2⌋ characters before and ⌈n/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  [tab:format.sign]

<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: 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:

double inf = numeric_limits<double>::infinity();
double nan = numeric_limits<double>::quiet_NaN();
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.
The width of a string is defined as the estimated number of column positions appropriate for displaying it in a terminal. [Note: This is similar to the semantics of the POSIX \texttt{wcswidth} function. — end note]

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: This is the case for Windows-based and many POSIX-based operating systems. — end note]

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:

- U+1100–U+115F
- U+2329–U+232A
- U+2E80–U+303E
- U+3040–U+A4CF
- U+AC00–U+D7A3
- U+F900–U+FAFF
- U+FE10–U+FE19
- U+FE30–U+FE6F
- U+FF00–U+FF60
- U+FFE0–U+FFE6
- U+1F300–U+1F64F
- U+1F900–U+1F9FF
- U+20000–U+2FFFD
- U+30000–U+3FFFD

The estimated width of other code points is 1.

For a string in a non-Unicode encoding, the width of a string is unspecified.

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 for arithmetic types other than \texttt{charT} and \texttt{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:

```cpp
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]

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.

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.

- For integral types, the locale-specific form causes the context’s locale to be used to insert the appropriate digit group separator characters.
- 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.
- For the textual representation of \texttt{bool}, the locale-specific form causes the context’s locale to be used to insert the appropriate string as if obtained with \texttt{numpunct::truename} or \texttt{numpunct::falsename}.

The type determines how the data should be presented.
The available string presentation types are specified in Table 61.

Table 61: Meaning of type options for strings [tab:format.type.string]

<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>

The meaning of some non-string presentation types is defined in terms of a call to to_chars. In such cases, let \([\text{first}, \text{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. \([\text{Note}: \text{Additional padding and adjustments are performed prior to copying the output through the output iterator as specified by the format specifiers. — end note}]

The available integer presentation types for integral types other than bool and charT are specified in Table 62.

[Example:

string s0 = format("{}"); 42); // value of s0 is "42"
string s1 = format("{0:b} {0:d} {0:o} {0:x}"; 42); // value of s1 is "101010 42 52 2a"
string s2 = format("{0:#x} {0:#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]

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>
</tbody>
</table>
| none               | The same as d. \([\text{Note}: \text{If the formatting argument type is charT or bool, the default is instead c or s, respectively. — end note}]

The available charT presentation types are specified in Table 63.

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>

The available bool presentation types are specified in Table 64.

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>

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,
Table 65: Meaning of type options for floating-point types  

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| a    | 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.  
| A    | The same as a, except that it uses uppercase letters for digits above 9 and P to indicate the exponent.  
| e    | 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.  
| E    | The same as e, except that it uses E to indicate exponent.  
| f, F | 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.  
| g    | 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.  
| G    | The same as g, except that it uses E to indicate exponent.  
| none | 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.  

respectively. For upper-case presentation types, infinity and NaN are formatted as INF and NAN, respectively.  
[Note: In either case, a sign is included if indicated by the sign option. — end note]  

The available pointer presentation types and their mapping to `to_chars` are specified in Table 66.  
[Note: Pointer presentation types also apply to `nullptr_t`. — end note]  

Table 66: Meaning of type options for pointer types  

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| none | If `uintptr_t` is defined,  
      | `to_chars(first, last, reinterpret_cast<uintptr_t>(value), 16)`  
      | with the prefix 0x added to the output; otherwise, implementation-defined.  
| p    | If `uintptr_t` is defined,  
      | `to_chars(first, last, reinterpret_cast<uintptr_t>(value), 16)`  
      | with the prefix 0x added to the output; otherwise, implementation-defined.  

20.20.3 Error reporting  

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 16.5.5.13.

20.20.4 Formatting 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.

```cpp
template<class... Args>
string format(string_view fmt, const Args&... args);
```

Effects: Equivalent to:

```cpp
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
— charT be decltype(fmt)::value_type,
— 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
— 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 + N). 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:

— it meets the

 REQUIRES(F::requires()

1 20.20.5.1 730
— Cpp17CopyMoveAssignable (Table 31), and
— Cpp17Destructible (Table 32)
requirements,
— it is swappable (16.5.3.2) for lvalues, and
— 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<chartT>,
(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() == '}'.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>f.parse(pc)</td>
<td>PC::iterator</td>
<td>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: 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.</td>
</tr>
<tr>
<td>f.format(t, fc)</td>
<td>FC::iterator</td>
<td>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).</td>
</tr>
<tr>
<td>f.format(u, fc)</td>
<td>FC::iterator</td>
<td>As above, but does not modify u.</td>
</tr>
</tbody>
</table>

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: 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:

(2.1) — The specializations
  template<> struct formatter<char, char>;
  template<> struct formatter<char, wchar_t>;
  template<> struct formatter<wchar_t, wchar_t>;

(2.2) — 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>;
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: 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<T, charT>, that specialization is enabled except as noted otherwise.

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_moveAssignable_v<F>.

6 An enabled specialization formatter<T, charT> meets the Formatter requirements (20.20.5.1). [Example:
   #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("{0}", 42); // OK, library-provided formatter
   std::string s1 = std::format("{0}", L"foo"); // error: disabled formatter
   std::string s2 = std::format("{0}", red);    // OK, user-provided formatter
   std::string s3 = std::format("{0}", err()); // error: disabled formatter
   —end example]

20.20.5.3 Class template basic_format_parse_context [format.parse.ctx]

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

        § 20.20.5.3 732
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);

};

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;

Effects: Initializes begin_ with fmt.begin(), end_ with fmt.end(), indexing_ with unknown, next_arg_id_ with 0, and num_args_ with num_args.

Returns: begin_.

constexpr const_iterator begin() const noexcept;

Returns: end_.

constexpr const_iterator end() const noexcept;

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 [format.context]

namespace std {
    template<class Out, class charT>
    class basic_format_context {
        basic_format_args<basic_format_context> args_;  // exposition only
    };

§ 20.20.5.4
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: 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]

```
§ 20.20.5.4 734
```
// 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"

20.20.6 Arguments

20.20.6.1 Class template basic_format_arg

namespace std {
    template<class Context>
    class basic_format_arg {
    public:
        using char_type = typename Context::char_type;
        // exposition only
        variant<monostate, bool, char_type,
        int, unsigned int, long long int, unsigned long long int,
        float, double, long double,
        const char_type*, basic_string_view<char_type>,
        const void*, handle> value;
        // exposition only

        template<class T> explicit basic_format_arg(const T& v) noexcept; // exposition only
        explicit basic_format_arg(float n) noexcept; // exposition only
        explicit basic_format_arg(double n) noexcept; // exposition only
        explicit basic_format_arg(long double n) noexcept; // exposition only
        explicit basic_format_arg(const char_type* s); // exposition only

        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;
    }
}

1 An instance of basic_format_arg provides access to a formatting argument for user-defined formatters.
2 The behavior of a program that adds specializations of basic_format_arg is undefined.
basic_format_arg() noexcept;

  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 (6.8.1) 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 (16.3.18).

  Effects: Initializes value with s.

template<class traits>
external basic_format_arg(basic_string_view<char_type, traits> s) noexcept;

  Effects: Initializes value with s.

template<class traits, class Allocator>
external basic_format_arg(
  const basic_string_view<char_type, traits, Allocator>& s) noexcept;

  Effects: Initializes value with basic_string_view<char_type>(s.data(), s.size()).

explicit basic_format_arg(nullptr_t) noexcept;

  Effects: Initializes value with static_cast<const void*>(nullptr).

template<class T> explicit basic_format_arg(const T* p) noexcept;

  Constraints: is_void_v<T> is true.

  Effects: Initializes value with p.

[Note: Constructing basic_format_arg from a pointer to a member is ill-formed unless the user provides an enabled specialization of formatter for that pointer to member type. — end note]
explicit operator bool() const noexcept;

Returns: \(!\text{holds_alternative<monostate>}(\text{value})\).

The class \texttt{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>;
    
        public:
            void format(basic_format_parse_context<char_type>&, Context& ctx) const;
    
    };
}
```

```cpp
template<class T> explicit handle(const T& val) noexcept;
```

Effects: Initializes \(\text{ptr}_\) with \texttt{addressof(val)} and \texttt{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));
}
```

```cpp
void format(basic_format_parse_context<char_type>& parse_ctx, Context& format_ctx) const;
```

Effects: Equivalent to: \texttt{format_}(\texttt{parse_ctx}, \texttt{format_ctx}, \texttt{ptr}_);

```cpp
template<class Visitor, class Context>
see below visit_format_arg(Visitor& vis, basic_format_arg<Context>& arg);
```

Effects: Equivalent to: \texttt{return visit(forward<Visitor>(vis), arg.value)};

### 20.20.6.2 Class template format-arg-store

```cpp
namespace std {
    template<class Context, class... Args>
    struct format-arg-store { // exposition only
        array<basic_format_arg<Context>, sizeof...(Args)> args;
    };
}
```

An instance of \texttt{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 \texttt{typename Context::template formatter_type<T,>} meets the \texttt{Formatter} requirements (20.20.5.1) for each \(T_i\) in \texttt{Args}.

Returns: \{basic_format_arg<Context>(\texttt{args})...\}.

```cpp
template<class... Args>
format-arg-store<wformat_context, Args...> make_wformat_args(const Args&... args);
```

Effects: Equivalent to: \texttt{return make_format_args(wformat_context)(\texttt{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
    };
}
```

§ 20.20.6.3
An instance of `basic_format_args` provides access to formatting arguments.

```cpp
basic_format_args() noexcept;
```

**Effects:** Initializes `size_` with `0`.

```cpp
template<class... Args>
  basic_format_args(const format_arg_store<Context, Args...>& store) noexcept;
```

**Effects:** Initializes `size_` with `sizeof...(Args)` and `data_` with `store.args.data()`.

```cpp
basic_format_arg<Context> get(size_t i) const noexcept;
```

**Returns:** `i < size_ ? data_[i] : basic_format_arg<Context>()`. 

[Note: 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.

```cpp
format_error(const string& what_arg);
```

**Postconditions:** `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
format_error(const char* what_arg);
```

**Postconditions:** `strcmp(what(), what_arg) == 0`. 

§ 20.20.7  738
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>
<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></td>
</tr>
<tr>
<td>21.4 String view classes</td>
<td>&lt;string_view&gt;</td>
</tr>
<tr>
<td>21.5 Null-terminated sequence utilities</td>
<td>&lt;cctype&gt;, &lt;cstdlib&gt;, &lt;cstring&gt;, &lt;cuchar&gt;, &lt;cwchar&gt;, &lt;cwctype&gt;</td>
</tr>
</tbody>
</table>

21.2 Character traits

This subclause 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. This subclause 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 (16.3.3) 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.

21.2.1 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>
<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>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::int_type</td>
<td></td>
<td>(described in 21.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::off_type</td>
<td></td>
<td>(described in 29.2.2 and 29.3)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::pos_type</td>
<td></td>
<td>(described in 29.2.2 and 29.3)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::state_type</td>
<td></td>
<td>(described in 21.2.2)</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>
<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>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</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::eof()</td>
<td>X::int_type</td>
<td>Returns: a value ( e ) such that ( X::eq_int_type(e, X::to_int_type(c)) ) 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.

### 21.2.2 Traits typedefs

```
using int_type = see below;
```

**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;
```

**Preconditions:** `state_type` meets the Cpp17Destructible (Table 32), Cpp17CopyAssignable (Table 31), Cpp17CopyConstructible (Table 29), and Cpp17DefaultConstructible (Table 27) requirements.

### 21.2.3 char_traits specializations

```cpp
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);
    }
}
```

*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>`.*

### 21.2.3.1 struct char_traits<char>

```cpp
namespace std {
    template<> struct char_traits<char> {
        // Definitions...
    }
}
```

2) 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`.

### 21.2.3.2 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;
    }
};
```

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-8 code unit.

### 21.2.3.3 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.

### 21.2.3.4 struct char_traits<char32_t>

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.

The member `eof()` returns an implementation-defined constant that cannot appear as a Unicode code point.

### 21.2.3.5 struct char_traits<wchar_t>

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 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 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 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 WEOF.

21.3 String classes

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.

21.3.1 Header <string> synopsis

#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.1

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 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);

§ 21.3.1
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 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 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+(const basic_string<charT, traits, Allocator>& lhs,
        charT rhs);

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)));

// 21.3.3.3, swap
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)));

// 21.3.3.4, 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);

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);

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);

// 21.3.3.5, erasure

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>;

// 21.3.4, 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(float val);
string to_string(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>;
}

// 21.3.5, 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 {
        // 21.3.6, suffix for basic_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);
    }
}

21.3.2 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 this Clause, 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:
                // types
                using traits_type = traits;

§ 21.3.2 747
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;

// 21.3.2.2, 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();
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>);

// 21.3.2.3, 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;

// 21.3.2.4, 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;

// 21.3.2.5, 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();

// 21.3.2.6, 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+=(initializer_list<charT>);
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,
    size_type n1, charT c);

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);

// 21.3.2 750

// 21.3.2.7, 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 rfnd(const T& t, size_type pos = npos) const noexcept(see below);
constexpr size_type rfnd(const basic_string& str, size_type pos = npos) const noexcept;
constexpr size_type rfnd(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfnd(const charT* s, size_type pos = npos) const;
constexpr size_type rfnd(charT c, size_type pos = npos) const noexcept;

template<class T>
constexpr size_type find_first_of(const T& t, size_type pos = 0) const noexcept(see below);
constexpr size_type find_first_of(const basic_string& str, 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 = npos) const;
constexpr size_type find_first_of(charT c, size_type pos = npos) const noexcept;

template<class T>
constexpr size_type find_last_of(const T& t, size_type pos = npos) const noexcept(see below);
constexpr size_type find_last_of(const basic_string& str, 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_last_of(charT c, size_type pos = npos) const noexcept;

template<class T>
constexpr size_type find_first_not_of(const T& t, size_type pos = 0) const noexcept(see below);
constexpr size_type find_first_not_of(const basic_string& str, 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_first_not_of(charT c, size_type pos = 0) const noexcept;

template<class T>
constexpr size_type find_last_not_of(const T& t, size_type pos = npos) const noexcept(see below);
constexpr size_type find_last_not_of(const basic_string& str, 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;
constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;

template<class T>
constexpr int compare(const T& t) const noexcept(see below);

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) const;

template<class T>
constexpr int compare(const basic_string& str) const noexcept;

template<class T>
constexpr int compare(size_type pos1, size_type n1, const basic_string& str) const;

template<class T>
constexpr int compare(size_type pos1, size_type n1, const basic_string& str, size_type pos2, size_type n2) const;

template<class T>
constexpr int compare(const charT* s) const;

template<class T>
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;

template<class T>
constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

constexpr basic_string substr(size_type pos = 0, size_type n = 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(const charT* x) const;
constexpr bool starts_with(const charT* x) const noexcept;
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;
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.

21.3.2.1 General requirements

If any operation would cause size() to exceed max_size(), that operation throws an exception object of type length_error.

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.

In every specialization basic_string<charT, traits, Allocator>, the type allocator_traits<Allocator>::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 specialization basic_string<charT, traits, Allocator>, the type traits shall meet the character traits requirements (21.2). [Note: The program is ill-formed if traits::char_type is not the same type as charT. —end note]

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:

1. Passing as an argument to any standard library function taking a reference to non-const basic_string as an argument.
2. Calling non-const member functions, except operator[], at, data, front, back, begin, rbegin, end, and rend.

21.3.2.2 Constructors and assignment operators

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.

---

222) For example, as an argument to non-member functions swap() (21.3.3.3), operator>>() (21.3.3.4), and getline() (21.3.3.4), or as an argument to basic_string::swap().
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());

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).
Postconditions: size() is equal to n, and traits::compare(data(), s, n) is equal to 0.

constexpr basic_string(const charT* s, const Allocator& a = Allocator());

Constraints: Allocator is a type that qualifies as an allocator (22.2.1). [Note: This affects class
template argument deduction. — end note]
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: This affects class
template argument deduction. — end note]
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<Input_iterator>::value_type, 

§ 21.3.2.2  753
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>;

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>;

Constraints: Allocator is a type that qualifies as an allocator (22.2.1).

constexpr 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.

constexpr basic_string& operator=(basic_string&& str)
noexcept(algorithm_traits<Allocator>::propagate_on_container_move_assignment::value ||
   algorithm_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>
constexpr 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);

constexpr basic_string& operator=(const charT* s);

Effects: Equivalent to: return *this = basic_string_view<charT, traits>(s);

constexpr basic_string& operator=(charT c);

Effects: Equivalent to:
    return *this = basic_string_view<charT, traits>(addressof(c), 1);

constexpr basic_string& operator=(initializer_list<charT> il);

Effects: Equivalent to:
    return *this = basic_string_view<charT, traits>(il.begin(), il.size());

21.3.2.3 Iterators support

constexpr iterator begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr 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()).

21.3.2.4 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 size_type capacity() const noexcept;

Returns: The size of the allocated storage in the string.
Complexity: Constant time.

constexpr size_type capacity() const noexcept;

Returns: The size of the allocated storage in the string.
Complexity: Constant time.

constexpr void resize(size_type n, charT c);

Effects: Alters the value of *this as follows:

— If \( n \leq \text{size()} \), erases the last \( \text{size()} - n \) elements.
— If \( n > \text{size()} \), appends \( n - \text{size()} \) 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: 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. [Note: If no reallocation happens, they remain valid. — end note]
constexpr void clear() noexcept;

Effects: Equivalent to: erase(begin(), end());

[[nodiscard]] constexpr bool empty() const noexcept;

Effects: Equivalent to: return size() == 0;

21.3.2.5 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);

Throws: out_of_range if pos >= size().

Returns: operator[](pos).

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);

21.3.2.6 Modifiers

21.3.2.6.1 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);

Effects: Equivalent to:

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);
21.3.2.6.2 basic_string::append

constexpr basic_string& append(const basic_string& str);

1 Effects: Equivalent to: return append(str.data(), str.size());

constexpr basic_string& append(const basic_string& str, size_type pos, size_type n = npos);

2 Effects: Equivalent to:
   return append(basic_string_view<charT, traits>(str).substr(pos, n));

template<class T>
constexpr basic_string& append(const T& t);

3 Constraints:
   — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
   — is_convertible_v<const T&, const charT*> is false.

4 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);

5 Constraints:
   — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
   — is_convertible_v<const T&, const charT*> is false.

6 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);

7 Preconditions: [s, s + n) is a valid range.

8 Effects: Appends a copy of the range [s, s + n) to the string.

9 Returns: *this.

constexpr basic_string& append(const charT* s);

10 Effects: Equivalent to: return append(s, traits::length(s));

constexpr basic_string& append(size_type n, charT c);

11 Effects: Appends n copies of c to the string.

12 Returns: *this.

template<class InputIterator>
constexpr basic_string& append(InputIterator first, InputIterator last);

13 Constraints: InputIterator is a type that qualifies as an input iterator (22.2.1).

14 Effects: Equivalent to: return append(basic_string(first, last, get_allocator()));

constexpr basic_string& append(initializer_list<charT> il);

15 Effects: Equivalent to: return append(il.begin(), il.size());

constexpr void push_back(charT c);

16 Effects: Equivalent to append(size_type{1}, c).

21.3.2.6.3 basic_string::assign

constexpr basic_string& assign(const basic_string& str);

1 Effects: Equivalent to: return *this = 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);

    Effects: Equivalent to: return *this = std::move(str);

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));

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.

    Effects: Equivalent to:
              basic_string_view<charT, traits> sv = t;
              return assign(sv.data(), sv.size());

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.

    Effects: Equivalent to:
              basic_string_view<charT, traits> sv = t;
              return assign(sv.substr(pos, n));

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.

constexpr basic_string& assign(const charT* s);

    Effects: Equivalent to: return assign(s, traits::length(s));

constexpr basic_string& assign(initializer_list<charT> il);

    Effects: Equivalent to: return assign(il.begin(), il.size());

constexpr basic_string& assign(size_type n, charT c);

    Effects: Equivalent to:
              clear();
              resize(n, c);
              return *this;

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()));

21.3.2.6.4 basic_string::insert [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:
          return insert(pos1, basic_string_view<charT, traits>(str), pos2, n);

template<class T>
constexpr basic_string& insert(size_type pos, const T& t);

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 insert(pos, sv.data(), sv.size());

template<class T>
constexpr basic_string& insert(size_type pos1, const T& t,
    size_type pos2, size_type n = npos);

Constraints:
(5.1)    — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
(5.2)    — is_convertible_v<const T&, const charT*>> is false.

Effects: Equivalent to:
          basic_string_view<charT, traits> sv = t;
          return insert(pos1, sv.substr(pos2, n));

constexpr basic_string& insert(size_type pos, const charT* s, size_type n);

Preconditions: [s, s + n) is a valid range.

Throws:
(8.1)    — out_of_range if pos > size(),
(8.2)    — length_error if n > max_size() - size(), or
(8.3)    — any exceptions thrown by allocator_traits<Alloc>::allocate.

Effects: Inserts a copy of the range [s, s + n) immediately before the character at position pos if
          pos < size(), or otherwise at the end of the string.

Returns: *this.

constexpr basic_string& insert(size_type pos, const charT* s);

Effects: Equivalent to: return insert(pos, s, traits::length(s));

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:
(14.1)    — out_of_range if pos > size(),
(14.2)    — length_error if n > max_size() - size(), or
(14.3)    — any exceptions thrown by allocator_traits<Alloc>::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.

customexpr iterator insert(const_iterator p, initializer_list<charT> il);

  Effects: Equivalent to: return insert(p, il.begin(), il.end());

21.3.2.6.5 basic_string::erase [string.erase]

customexpr basic_string& erase(size_type pos = 0, size_type n = npos);

  Throws: out_of_range if pos > size().
  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.

customexpr iterator erase(const_iterator p);

  Preconditions: p is a valid dereferenceable iterator on *this.
  Throws: Nothing.
  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.

customexpr iterator erase(const_iterator first, const_iterator last);

  Preconditions: first and last are valid iterators on *this. [first, last) is a valid range.
  Throws: Nothing.
  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.

customexpr void pop_back();

  Preconditions: !empty().
  Throws: Nothing.
  Effects: Equivalent to erase(end() - 1).

21.3.2.6.6 basic_string::replace [string.replace]

customexpr basic_string& replace(size_type pos1, size_type n1, const basic_string& str);

  Effects: Equivalent to: return replace(pos1, n1, str.data(), str.size());

customexpr 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);

3 Constraints:
— is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
— is_convertible_v<const T&, const charT*> is false.

4 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);

5 Constraints:
— is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
— is_convertible_v<const T&, const charT*> is false.

6 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);

7 Preconditions: [s, s + n2) is a valid range.

8 Throws:
— out_of_range if pos1 > size(),
— length_error if the length of the resulting string would exceed max_size() (see below), or
— any exceptions thrown by allocator_traits<Allocator>::allocate.

9 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 a copy of the range [s, s + n2).

Returns: *this.

constexpr basic_string& replace(size_type pos, size_type n, const charT* s);

11 Effects: Equivalent to: return replace(pos, n, s, traits::length(s));

constexpr basic_string& replace(size_type pos1, size_type n1, size_type n2, charT c);

12 Throws:
— out_of_range if pos1 > size(),
— length_error if the length of the resulting string would exceed max_size() (see below), or
— any exceptions thrown by allocator_traits<Allocator>::allocate.

13 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.

constexpr basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);

15 Effects: Equivalent to: return replace(i1, i2, basic_string_view<charT, traits>(str));

template<class T>
constexpr basic_string& replace(const_iterator i1, const_iterator i2, const T& t);

16 Constraints:
— is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
— is_convertible_v<const T&, const charT*> is false.

17 Preconditions: \[\text{begin()}, \text{i1}\text{)}\text{ and }\[\text{i1}, \text{i2}\text{)}\text{ are valid ranges.}\]

18 Effects: Equivalent to:

\[
\text{basic_string_view<charT, traits> sv = t;}\\
\text{return replace(}\text{i1 - begin()}, \text{i2 - i1, sv.data()}, \text{sv.size()});
\]

constexpr basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);

Effects: Equivalent to: return replace(i1, i2, basic_string_view<charT, traits>(s, n));

constexpr basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);

Effects: Equivalent to: return replace(i1, i2, basic_string_view<charT, traits>(s));

constexpr basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);

Preconditions: \[\text{begin()}, \text{i1}\text{)}\text{ and }\[\text{i1}, \text{i2}\text{)}\text{ are valid ranges.}\]

Effects: Equivalent to: return replace(i1 - begin(), i2 - i1, n, c);

template<class InputIterator>
constexpr 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: return replace(i1, i2, basic_string(j1, j2, get_allocator()));

constexpr basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);

Effects: Equivalent to: return replace(i1, i2, il.begin(), il.size());

21.3.2.6.7 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: This does not terminate s with a null object. — end note]

21.3.2.6.8 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.

21.3.2.7 String operations

21.3.2.7.1 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, \text{size}\text{)}\text{.}

Complexity: Constant time.

Remarks: The program shall not modify any of the values stored in the character array; otherwise, the
behavior is undefined.

constexpr charT* data() noexcept;

Returns: A pointer p such that p + i == addressof(operator[])(i) for each i in \[0, \text{size}\text{)}\text{.}

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.

\[
\text{constexpr operator basic_string_view<charT, traits>() const noexcept;}
\]

Effects: Equivalent to: \( \text{return basic_string_view<charT, traits>(data(), size());} \)

\[
\text{constexpr allocator_type get_allocator() const noexcept;}
\]

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.

### 21.3.2.7.2 Searching

Let \( F \) be one of \( \text{find, rfind, find_first_of, find_last_of, find_first_not_of, and find_last_not_of.} \)

1. Each member function of the form
   \[
   \text{constexpr size_type } F(\text{const basic_string& str, size_type pos}) \text{ const noexcept;}
   \]
   has effects equivalent to: \( \text{return } F(\text{basic_string_view<charT, traits>(str), pos);} \)

2. Each member function of the form
   \[
   \text{constexpr size_type } F(\text{const charT* s, size_type pos}) \text{ const;}
   \]
   has effects equivalent to: \( \text{return } F(\text{basic_string_view<charT, traits>(s), pos);} \)

3. Each member function of the form
   \[
   \text{constexpr size_type } F(\text{const charT* s, size_type pos, size_type n}) \text{ const;}
   \]
   has effects equivalent to: \( \text{return } F(\text{basic_string_view<charT, traits>(s, n), pos);} \)

4. Each member function of the form
   \[
   \text{constexpr size_type } F(\text{charT c, size_type pos}) \text{ const noexcept;}
   \]
   has effects equivalent to:
   \[
   \text{return } F(\text{basic_string_view<charT, traits>(addressof(c), 1), pos);} \)

#### Constraints:

1. \( \text{is_convertible_v<const T&, basic_string_view<charT, traits>>} \) is true and
2. \( \text{is_convertible_v<const T&, const charT*>>} \) is false.

Effects: Let \( G \) be the name of the function. Equivalent to:
   \[
   \text{basic_string_view<charT, traits> } s = *\text{this, sv = t;}
   \]
   \[
   \text{return s.}G(\text{sv, pos);} \)

Remarks: The expression inside noexcept is equivalent to \( \text{is_nothrow_convertible_v<const T&, basic_string_view<charT, traits>>} \).

### 21.3.2.7.3 basic_string::substr

\[
\text{constexpr basic_string substr(size_type pos = 0, size_type n = npos) const;}
\]

Throws: \( \text{out_of_range} \) if \( \text{pos} > \text{size()} \).

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).

21.3.2.7.4 basic_string::compare

template<class T>
constexpr int compare(const T& t) const noexcept(see below);

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);

21.3.2.7.5 basic_string::starts_with

constexpr bool starts_with(basic_string_view<charT, traits> x) const noexcept;
constexpr bool starts_with(charT x) const noexcept;

§ 21.3.2.7.5
constexpr bool starts_with(const charT* x) const;

Effects: Equivalent to:

    return basic_string_view<charT, traits>(data(), size()).starts_with(x);

21.3.2.7.6 basic_string::ends_with

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:

    return basic_string_view<charT, traits>(data(), size()).ends_with(x);

21.3.3 Non-member functions

21.3.3.1 operator+

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+(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Effects: Equivalent to:

    basic_string<charT, traits, Allocator> r = lhs;
    r.append(rhs);
    return r;

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:

    lhs.append(rhs);
    return std::move(lhs);

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:

    rhs.insert(0, lhs);
    return std::move(rhs);

except that both lhs and rhs are left in valid but unspecified states. [Note: If lhs and rhs have equal allocators, the implementation may move from either. —end note]

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);

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+(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+(const 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, 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);

21.3.3.2 Non-member comparison 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 charT lhs, const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
constexpr basic_string_view<charT, traits><(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs) noexcept;

template<class charT, class traits, class Allocator>
constexpr basic_string_view<charT, traits><(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs) noexcept;

Effects: Let op be the operator. Equivalent to:
return basic_string_view<charT, traits>(lhs) op basic_string_view<charT, traits>(rhs);
21.3.3.3 swap

template<class charT, class traits, class Allocator>
constexpr void
swap(basic_string<charT, traits, Allocator>& lhs,
    basic_string<charT, traits, Allocator>& rhs);

Effects: Equivalent to lhs.swap(rhs).

21.3.3.4 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);

Effects: Behaves as a formatted input function (29.7.4.2.1). 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);

Effects: Equivalent to: return os << basic_string_view<charT, traits>(str);

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
ggetline(basic_istream<charT, traits>& is, basic_string<charT, traits, Allocator>& str, charT delim);

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
ggetline(basic_istream<charT, traits>&& is, basic_string<charT, traits, Allocator>& str, charT delim);

Effects: Behaves as an unformatted input function (29.7.4.3), 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)
3. 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).
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
generate(basic_istream<charT, traits>&& is,  
           basic_string<charT, traits, Allocator>& str);

Returns: is.

template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
generate(basic_istream<charT, traits>&& is,  
           basic_string<charT, traits, Allocator>& str);

Returns: generate(is, str, is.widen('\n')).

21.3.3.5 Erasure

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;

21.3.4 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), stoll(str.c_str(), ptr, base), and stoull(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 != 0, 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 stoull reports that no conversion  
could be performed. Throws out_of_range if strtol, stoul, stoll or stoull 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 strtof(  
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 != 0, 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(float val);
string to_string(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", "%e", 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` != 0, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: The converted result.

```
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` != 0, the function stores in `*idx` the index of the first unconverted element of `str`.

Returns: The converted result.

```
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(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"%e", or L"%Lf", respectively, where `buf` designates an internal character buffer of sufficient size `buffsz`.

§ 21.3.4 769
21.3.5 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 \( \text{hash}<S>()(s) == \text{hash}<SV>()(SV(s)) \).

21.3.6 Suffix for basic_string literals

```
constexpr string operator"s(const char* str, size_t len);
```

1 Returns: `string{str, len}`.

```
constexpr u8string operator"s(const char8_t* str, size_t len);
```

2 Returns: `u8string{str, len}`.

```
constexpr u16string operator"s(const char16_t* str, size_t len);
```

3 Returns: `u16string{str, len}`.

```
constexpr u32string operator"s(const char32_t* str, size_t len);
```

4 Returns: `u32string{str, len}`.

```
constexpr wstring operator"s(const wchar_t* str, size_t len);
```

5 Returns: `wstring{str, len}`.

6 [Note: The same suffix \( s \) is used for \texttt{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

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 this subclause, the type of the char-like objects held in a `basic_string_view` object is designated by `charT`.

2 [Note: The library provides implicit conversions from `const charT*` 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 should define their own implicit conversions to `std::basic_string_view` in order to interoperate with these functions. — end note]

21.4.1 Header `<string_view>` synopsis

```
#include <compare>
// see 17.11.1

namespace std {
  // 21.4.2, class template basic_string_view
  template<class charT, class traits = char_traits<charT>>
  class basic_string_view;

  template<class charT, class traits>
  inline constexpr bool ranges::enable_view<basic_string_view<charT, traits>> = true;

  template<class charT, class traits>
  inline constexpr bool ranges::enable_borrowed_range<basic_string_view<charT, traits>> = true;
}
```

§ 21.4.1 770
// 21.4.4, non-member comparison functions
template<class charT, class traits>
constexpr bool operator==(basic_string_view<charT, traits> x,
                         basic_string_view<charT, traits> y) noexcept;

template<class charT, class traits>
constexpr
  see below operator<=>(basic_string_view<charT, traits> x,
                       basic_string_view<charT, traits> y) noexcept;

// see 21.4.4, sufficient additional overloads of comparison functions

// 21.4.5, inserters and extractors
template<class charT, class traits>
basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os,
            basic_string_view<charT, traits> str);

using string_view = basic_string_view<char>;
using u8string_view = basic_string_view<char8_t>;
using u16string_view = basic_string_view<char16_t>;
using u32string_view = basic_string_view<char32_t>;
using wstring_view = basic_string_view<wchar_t>;

// 21.4.6, hash support
template<class T> struct hash;
template<> struct hash<string_view>;
template<> struct hash<u8string_view>;
template<> struct hash<u16string_view>;
template<> struct hash<u32string_view>;
template<> struct hash<wstring_view>;

inline namespace literals {
  inline namespace string_view_literals {
    constexpr string_view operator"sv(const char* str, size_t len) noexcept;
  }
}

1 The function templates defined in 20.2.2 and 23.7 are available when <string_view> is included.

21.4.2 Class template basic_string_view

```cpp
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 21.4.2.2
    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;
```

223] Because basic_string_view refers to a constant sequence, iterator and const_iterator are the same type.
static constexpr size_type npos = size_type(-1);

// 21.4.2.1, 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);

// 21.4.2.2, 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;

// 21.4.2.3, 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;

// 21.4.2.4, 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;

// 21.4.2.5, modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_string_view& s) noexcept;

// 21.4.2.6, 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;
constexpr bool ends_with(const charT* x) const;

// 21.4.2.7, 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* a, size_type pos, size_type n) const;
constexpr size_type find(const charT* a, 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 rfind_first_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type rfind_first_of(const charT* s, size_type pos = 0) const noexcept;
constexpr size_type rfind_last_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind_last_of(const charT* s, size_type pos = npos) const noexcept;
constexpr size_type rfind_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type rfind_first_not_of(const charT* s, size_type pos = 0) const noexcept;
constexpr size_type rfind_last_not_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind_last_not_of(const charT* s, size_type pos = npos) const noexcept;

private:
    const_pointer data_; // exposition only
    size_type size_; // exposition only
};

// 21.4.3, 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: 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.

21.4.2.1 Construction and assignment [string.view.cons]
constexpr basic_string_view() noexcept;
Postconditions: size_ == 0 and data_ == nullptr.

constexpr basic_string_view(const charT* str);
Preconditions: [str, str + traits::length(str)) is a valid range.
Effects: Constructs a basic_string_view, initializing data_ with str and initializing size_ with traits::length(str).
Complexity: \(O(\text{traits::length}(\text{str}))\).

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.
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.

21.4.2.2 Iterator support

using const_iterator = implementation-defined;

A type that meets the requirements of a constant Cpp17RandomAccessIterator (23.3.5.6), 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()).

21.4.2.3 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.

21.4.2.4 Element access

constexpr const_reference operator[](size_type pos) const;

Preconditions: pos < size().

Returns: data_[pos].

Throws: Nothing.
[Note: 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;
   Throws: out_of_range if pos >= size().
   Returns: data_[pos].

constexpr const_reference front() const;
   Preconditions: !empty().
   Returns: data_[0].
   Throws: Nothing.

constexpr const_reference back() const;
   Preconditions: !empty().
   Returns: data_[size() - 1].
   Throws: Nothing.

constexpr const_pointer data() const noexcept;
   Returns: data_.

[Note: Unlike basic_string::data() and string-literals, data() may 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]

21.4.2.5 Modifiers

constexpr void remove_prefix(size_type n);
   Preconditions: n <= size().
   Effects: Equivalent to: data_ += n; size_ -= n;

constexpr void remove_suffix(size_type n);
   Preconditions: n <= size().
   Effects: Equivalent to: size_ -= n;

constexpr void swap(basic_string_view& s) noexcept;
   Effects: Exchanges the values of *this and s.

21.4.2.6 String operations

constexpr size_type copy(charT* s, size_type n, size_type pos = 0) const;
   Let rlen be the smaller of n and size() - pos.
   Throws: out_of_range if pos > size().
   Preconditions: [s, s + rlen) is a valid range.
   Effects: Equivalent to traits::copy(s, data() + pos, rlen).
   Returns: rlen.
   Complexity: $O(rlen)$.

constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;
   Let rlen be the smaller of n and size() - pos.
   Throws: out_of_range if pos > size().
   Effects: Determines rlen, the effective length of the string to reference.
   Returns: basic_string_view(data() + pos, rlen).
constexpr int compare(basic_string_view str) const noexcept;

Let \( rlen \) be the smaller of \( \text{size()} \) and \( \text{str.size()} \).

**Effects:** Determines \( rlen \), the effective length of the strings to compare. The function then compares the two strings by calling \( \text{traits::compare(data()), str.data(), rlen} \).

**Complexity:** \( O(rlen) \).

**Returns:** The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 70.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{size()} &lt; \text{str.size()} )</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>( \text{size()} == \text{str.size()} )</td>
<td>0</td>
</tr>
<tr>
<td>( \text{size()} &gt; \text{str.size()} )</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>

constexpr int compare(size_type pos1, size_type n1, basic_string_view str) const;

**Effects:** Equivalent to: return \( \text{substr(pos1, n1).compare(str)} \);

constexpr int compare(size_type pos1, size_type n1, basic_string_view str, size_type pos2, size_type n2) const;

**Effects:** Equivalent to: return \( \text{substr(pos1, n1).compare(str.substr(pos2, n2))} \);

constexpr int compare(const charT* s) const;

**Effects:** Equivalent to: return \( \text{compare(basic_string_view(s))} \);

constexpr int compare(size_type pos1, size_type n1, const charT* s) const;

**Effects:** Equivalent to: return \( \text{substr(pos1, n1).compare(basic_string_view(s))} \);

constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

**Effects:** Equivalent to: return \( \text{substr(pos1, n1).compare(basic_string_view(s, n2))} \);

constexpr bool starts_with(basic_string_view x) const noexcept;

**Effects:** Equivalent to: return \( \text{substr(0, x.size()).compare(x)} \);

constexpr bool starts_with(charT x) const noexcept;

**Effects:** Equivalent to: return \( \text{!empty() \&\& traits::eq(front(), x)} \);

constexpr bool starts_with(const charT* x) const;

**Effects:** Equivalent to: return \( \text{starts_with(basic_string_view(x))} \);

constexpr bool ends_with(basic_string_view x) const noexcept;

**Effects:** Equivalent to: return \( \text{size() \geq x.size() \&\& compare(size() - x.size(), npos, x) == 0} \);

constexpr bool ends_with(charT x) const noexcept;

**Effects:** Equivalent to: return \( \text{!empty() \&\& traits::eq(back(), x)} \);

constexpr bool ends_with(const charT* x) const;

**Effects:** Equivalent to: return \( \text{ends_with(basic_string_view(x))} \);

### 21.4.2.7 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 \( \text{find, rfind, find_first_of, find_last_of, find_first_not_of, and find_last_not_of} \).

---

(2.1) Each member function of the form
constexpr return-type F(const charT* s, size_type pos) const;

has effects equivalent to: return F(basic_string_view(s), pos);

(2.2) — Each member function of the form
        constexpr return-type F(const charT* s, size_type pos, size_type n) const;
        has effects equivalent to: return F(basic_string_view(s, n), pos);

(2.3) — Each member function of the form
        constexpr return-type F(charT c, size_type pos) const noexcept;
        has effects equivalent to: return F(basic_string_view(addressof(c), 1), pos);

customexpr size_type find(basic_string_view str, size_type pos = 0) const noexcept;

Let xpos be the lowest position, if possible, such that the following conditions hold:

- pos <= xpos
- xpos + str.size() <= size()
- traits::eq(at(xpos + I), str.at(I)) for all elements 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.

customexpr size_type rfind(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 + str.size() <= size()
- traits::eq(at(xpos + I), str.at(I)) for all elements 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.

customexpr size_type find_first_of(basic_string_view str, size_type pos = 0) const noexcept;

Let 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 some 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.

customexpr size_type find_last_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 some 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.

customexpr size_type find_first_not_of(basic_string_view str, size_type pos = 0) const noexcept;

Let 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.

### 21.4.3 Deduction guide

```
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>.

### 21.4.4 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 71: Additional basic_string_view comparison overloads

<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: A sample conforming implementation for operator== would be:
```
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;
  }
```
]
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 see 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).

21.4.5 Insetters 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 (29.7.5.2.1) 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 29.7.5.2.1. 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

21.4.6 Hash support

The specialization is enabled (20.14.18). [Note: The hash value of a string view object is equal to the hash value of the corresponding string object (21.3.5). — end note]

21.4.7 Suffix for basic_string_view literals

constexpr string_view operator"sv(const char* str, size_t len) noexcept;
Returns: string_view(str, len).

constexpr u8string_view operator"sv(const char8_t* str, size_t len) noexcept;
Returns: u8string_view(str, len).

constexpr u16string_view operator"sv(const char16_t* str, size_t len) noexcept;
Returns: u16string_view(str, len).

constexpr u32string_view operator"sv(const char32_t* str, size_t len) noexcept;
Returns: u32string_view(str, len).

constexpr 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);
  int iscntrl(int c);
  int isdigit(int c);
  int isgraph(int c);
  int islower(int c);
}
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
}

1 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 iswpunct(wint_t wc);
    int iswspace(wint_t wc);
    int iswupper(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);
}
```

#define WEOF see below

1 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);
    char* strncat(char* s1, const char* s2, size_t n);
    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);
    size_t strxfrm(char* s1, const char* s2, size_t n);
    int memchr(const char* s1, const char* s2, size_t n);
    int memcmp(const void* s1, const void* s2, size_t n);
    int memchr(void* s1, int c, size_t n);
    char* strchr(const char* s1, int c);
    char* strchr(char* s1, int c);
    size_t strcspn(const char* s1, const char* s2);
    }
```

§ 21.5.3 780
const char* strpbrk(const char* s1, const char* s2); // see 16.2
char* strpbrk(char* s1, const char* s2); // see 16.2
const char* strrchr(const char* s, int c); // see 16.2
char* strrchr(char* s, int c); // see 16.2
size_t strspn(const char* s1, const char* s2); // see 16.2
const char* strstr(const char* s1, const char* s2); // see 16.2
char* strstr(char* s1, const char* s2); // see 16.2
size_t strlen(const char* s);

#define NULL see 17.2.3

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 (16.5.5.10).

The functions memcpy and memmove are signal-safe (17.13.4). 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: 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

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 swscanf(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 vswprintf(wchar_t* s, size_t n, const wchar_t* format, va_list arg);
  int vswscanf(const wchar_t* s, const wchar_t* format, va_list arg);
  int wprintf(const wchar_t* format, va_list arg);
  int wscanf(const wchar_t* format, ...);
  wchar_t* wcscpy(wchar_t* s1, const wchar_t* s2);
  wchar_t* wcsncpy(wchar_t* s1, const wchar_t* s2, size_t n);

  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);
  int putwchar(wchar_t c);
  int ungetwc(wint_t c, FILE* stream);
  wchar_t* wcsxfrm(const wchar_t* s1, const wchar_t* s2, size_t n);

[21.5.4]
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`.

[Note: The functions `wcschr`, `wcsrchr`, `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]
The contents and meaning of the header \texttt{<uchar>}
are the same as the C standard library header \texttt{<uchar.h>},
extunless it declares the additional \texttt{mbrtoc8} and \texttt{c8rtomb} functions and does not declare types \texttt{char16_t} nor \texttt{char32_t}.

\textbf{See also:} ISO C 7.28

\subsection*{21.5.6 Multibyte / wide string and character conversion functions} \[\texttt{<cuchar.h>}\]

\footnote{Note: The headers \texttt{<cstdlib>}, \texttt{<cuchar>} and \texttt{<cwchar>} declare the functions described in this subclause. — end note}

\begin{verbatim}
int mbsinit(const mbstate_t* ps);
int mblen(const char* s, size_t n);
size_t mbstowcs(wchar_t* pcws, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pcws, size_t n);
\end{verbatim}

\textit{Effects:} These functions have the semantics specified in the C standard library.

\textbf{See also:} ISO C 7.22.7.1, 7.22.8, 7.29.6.2.1

\begin{verbatim}
int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);
\end{verbatim}

\textit{Effects:} These functions have the semantics specified in the C standard library.

\textbf{Remarks:} Calls to these functions may introduce a data race (16.5.5.10) with other calls to the same function.

\textbf{See also:} ISO C 7.22.7

\begin{verbatim}
size_t mbrlen(const char* s, size_t n, mbstate_t* ps);
size_t mbtowc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);
size_t wcrtomb(char* s, wchar_t wchar, mbstate_t* ps);
\end{verbatim}

\textit{Effects:} These functions have the semantics specified in the C standard library.

\textbf{Remarks:} Calling these functions with an \texttt{mbstate_t*} argument that is a null pointer value may introduce a data race (16.5.5.10) with other calls to the same function.

\textbf{See also:} ISO C 7.29.6.3

\begin{verbatim}
size_t mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);
\end{verbatim}

\textit{Effects:} If \texttt{s} is a null pointer, equivalent to \texttt{mbrtoc8(nullptr, ",", 1, ps)}. Otherwise, the function inspects at most \texttt{n} bytes beginning with the byte pointed to by \texttt{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 \texttt{pc8} is not a null pointer, stores the value of the first (or only) such code unit in the object pointed to by \texttt{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.

\textbf{Returns:} The first of the following that applies (given the current conversion state):

\begin{enumerate}
\item \texttt{0}, if the next \texttt{n} or fewer bytes complete the multibyte character that corresponds to the U+0000 Unicode character (which is the value stored).
\item \texttt{1} to \texttt{n} (inclusive), if the next \texttt{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.
\item \texttt{(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).
\item \texttt{(size_t)(-2)}, if the next \texttt{n} bytes contribute to an incomplete (but potentially valid) multibyte character, and all \texttt{n} bytes have been processed (no value is stored).
\end{enumerate}
— (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.

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, (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 (16.5.5.10) 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: 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 (20.10.9.2), 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: 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</td>
<td>T</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</td>
<td>T&amp;</td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_reference</td>
<td>const T&amp;</td>
<td></td>
<td></td>
<td>compile time</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></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::const_iterator</td>
<td>constant</td>
<td>iterator type</td>
<td>any iterator category</td>
<td>compile time</td>
</tr>
<tr>
<td></td>
<td>iterator type</td>
<td>whose value type is T</td>
<td>that meets the</td>
<td>compile time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>forward iterator</td>
<td>compile time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>requirements.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integer</td>
<td>type</td>
<td>is identical to the</td>
<td>compile time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>difference type of</td>
<td>compile time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X::iterator and</td>
<td>compile time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>X::const_iterator</td>
<td>compile time</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></td>
<td></td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X u;</td>
<td>constant</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>X()</td>
<td>constant</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>X(a)</td>
<td>constant</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>X u(a); X u = a;</td>
<td>linear</td>
<td></td>
<td>Preconditions: T is Cpp17CopyInsertable into X (see below). Postconditions: a == X(a).</td>
<td>linear</td>
</tr>
<tr>
<td>X u(rv); X u = rv;</td>
<td>linear</td>
<td></td>
<td>Postconditions: u is equal to the value that rv had before this construction</td>
<td>linear</td>
</tr>
<tr>
<td>a = rv X&amp;</td>
<td>linear</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>
</tr>
<tr>
<td>a.~X()</td>
<td>void</td>
<td>Effects: destroys every element of a; any memory obtained is deallocated.</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a.begin()</td>
<td>iterator;</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>const_iterator</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>iterator for</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>constant a</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.end()</td>
<td>iterator;</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>const_iterator</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>iterator for</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>constant a</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.cbegin()</td>
<td>const_iterator</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>const_cast&lt;X</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>const&amp;&gt;(a)</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>.begin();</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.cend()</td>
<td>const_iterator</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>const_cast&lt;X</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>const&amp;&gt;(a).end();</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
</tbody>
</table>

§ 22.2.1 786
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>i &lt;=&gt; j</td>
<td>strong_ ordering</td>
<td>Constraints: X::iterator meets the random access iterator requirements.</td>
<td>constant</td>
<td></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>Effects: exchanges the contents of a and b</td>
<td>linear</td>
</tr>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td>Equivalent to a.swap(b)</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td>swap(a, b)</td>
<td>void</td>
<td>Postconditions: r == a</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td>r = a</td>
<td>X&amp;</td>
<td>a.size() distance(a.begin(), a.end())</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.max_size() distance(b.begin(), b.end()) for the largest possible container</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.empty()</td>
<td>convertible to bool</td>
<td>a.begin() == a.end()</td>
<td>constant</td>
<td></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: 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 16.5.3.5). [Note: In particular, containers and iterators do not store references to allocated elements other than through the allocator’s pointer type, i.e., as objects of type P or pointer_traits<P>::template rebind<U> where P is allocator_traits<allocator_type>::pointer. — end note] 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.

§ 22.2.1
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. [Note: If an invocation of a constructor uses the default value of an optional allocator argument, then the allocator type must support value-initialization. — end note] 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 `allocator_traits<allocator_type>::propagate_on_container_copy_assignment::value`, `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value`, or `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.

9 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. Values 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 16.5.3.2. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is true, then values of type allocator type shall be swappable and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in 16.5.3.2. 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.

10 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 pre-/post-condition</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_reverse_iterator&gt;</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>a.rbegin()</code></td>
<td><code>reverse_iterator</code>; <code>const_reverse_iterator</code> for <code>a</code></td>
<td><code>reverse_iterator&lt;end()&gt;</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>a.rend()</code></td>
<td><code>reverse_iterator</code>; <code>const_reverse_iterator</code> for <code>a</code></td>
<td><code>reverse_iterator&lt;begin()&gt;</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>a.crbegin()</code></td>
<td><code>const_reverse_iterator</code></td>
<td><code>const_cast&lt;X const&amp;&gt;(a).rbegin()</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>a.crend()</code></td>
<td><code>const_reverse_iterator</code></td>
<td><code>const_cast&lt;X const&amp;&gt;(a).rend()</code></td>
<td>constant</td>
</tr>
</tbody>
</table>

11 Unless otherwise specified (see 22.2.6.1, 22.2.7.1, 22.3.8.4, and 22.3.11.5) all container types defined in this Clause meet the following additional requirements:

11.1 — if an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

11.2 — if an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

11.3 — no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

11.4 — no copy constructor or assignment operator of a returned iterator throws an exception.

11.5 — no `swap()` function throws an exception.
— no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: The `end()` iterator does not refer to any element, so it may be invalidated. — end note]

12 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 (23.3.5.6) 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>
<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><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><code>Preconditions: Either &lt;= is defined for values of type (possibly const) T, or &lt; is defined for values of type (possibly const) T and &lt; is a total ordering relationship.</code></td>
<td>linear</td>
</tr>
</tbody>
</table>

[Note: The algorithm `lexicographical_compare_three_way` is defined in Clause 25. — end note]

15 All of the containers defined in this Clause and in 21.3.2 except `array` meet the additional requirements of an allocator-aware container, as described in Table 76.

Given an allocator type `A` and given a container type `X` having a `value_type` identical to `T` and an `allocator_type` identical to `allocator_traits<A>::rebind_alloc<T>` and given an `lvalue 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:

(15.1) — `T` is `Cpp17DefaultInsertable into X` means that the following expression is well-formed:

`allocator_traits<A>::construct(m, p)`

(15.2) — 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`.

(15.3) — `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: `rv` remains a valid object. Its state is unspecified — end note]

(15.4) — `T` is `Cpp17CopyInsertable into X` means that, in addition to `T` being `Cpp17MoveInsertable into X`, the following expression is well-formed:

`allocator_traits<A>::construct(m, p, v)`

and its evaluation causes the following postcondition to hold: The value of `v` is unchanged and is equivalent to `*p`.

(15.5) — `T` is `Cpp17EmplaceConstructible into X from args`, for zero or more arguments `args`, means that the following expression is well-formed:

`allocator_traits<A>::construct(m, p, args)`
(15.6) — T is \textit{C++17Erasable from X} means that the following expression is well-formed:

\[
\text{allocator_traits<A>::destroy(m, p)}
\]

\[\text{Note: A container calls allocator_traits\textlangle A\textrangle::construct(m, p, args) to construct an element at p using args, with m == get_allocator(). The default construct in allocator will call \textbar new((void*)p) T(args), but specialized allocators may choose a different definition. — \textit{end note}}\]

In Table 76, X denotes an allocator-aware container class with a \textit{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 76: Allocator-aware container requirements  

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
</table>
| allocator_\text-
\text{-type} A | M\textit{andates:} allocator_type::value_type is the same as X::value_type. | compile time |
| get_\text-
\text{-allocator()} A | | constant |
| X() \newline X u(); | Preconditions: A meets the \textit{Cpp17DefaultConstructible} requirements. \newline Postconditions: u.empty() returns true, u.get_allocator() == A() | constant |
| X(m) \newline X u(m); | Postconditions: u.empty() returns true, u.get_allocator() == m | constant |
| X(t, m) \newline X u(t, m); | Preconditions: T is \textit{Cpp17CopyInsertable} into X. \newline Postconditions: u == t, u.get_allocator() == m | linear |
| X(rv) \newline X u(rv); | Postconditions: u has the same elements as rv had before this construction; the value of u.get_allocator() is the same as the value of rv.get_allocator() before this construction. | constant |
| X(rv, m) \newline X u(rv, m); | Preconditions: T is \textit{Cpp17MoveInsertable} into X. \newline Postconditions: u has the same elements, or copies of the elements, that rv had before this construction, u.get_allocator() == m | constant if m == rv.get_-\text{-allocator()}, otherwise linear |
| a = t \newline X& | Preconditions: T is \textit{Cpp17CopyInsertable} into X and \textit{Cpp17CopyAssignable}. \newline Postconditions: a == t | linear |
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>a = rv X&amp;</td>
<td>X&amp;</td>
<td>Preconditions: If allocator_traits&lt;allocator_type&gt;::propagate_on_container_move_assignment::value is false, T is Cpp17MoveInsertable into X and Cpp17MoveAssignabile. Effects: All existing elements of a are either move assigned to or destroyed. Postconditions: a is equal to the value that rv had before this assignment.</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>constant</td>
</tr>
</tbody>
</table>

17 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:

(17.1) — The qualified-id A::value_type is valid and denotes a type (13.10.2).

(17.2) — The expression declval<A&>().allocate(size_t{}) is well-formed when treated as an unevaluated operand.

22.2.2 Container data races [container.requirements.dataraces]

1 For purposes of avoiding data races (16.5.5.10), 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[].

2 Notwithstanding 16.5.5.10, 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.

3 [Note: 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 may result in a data race. As an exception to the general rule, for a vector<bool> y, y[0] = true may race with y[1] = true. — end note]

22.2.3 Sequence containers [sequence.reqmts]

1 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 might define).

2 [Note: The sequence containers offer the programmer different complexity trade-offs and should be used accordingly. vector is the type of sequence container that should be used by default. array should be used when the container has a fixed size known during translation. list or forward_list should be used when there are frequent insertions and deletions from the middle of the sequence. deque is the data structure of choice when most insertions and deletions take 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 \( X::\text{allocator\_type} \) if the qualified-id \( X::\text{allocator\_type} \) is valid and denotes a type (13.10.2) and \( \text{allocator}\langle T \rangle \) if it doesn’t, \( i \) and \( j \) denote iterators that meet the \( \text{Cpp17InputIterator} \) requirements and refer to elements implicitly convertible to \( \text{value\_type} \), \( [i, j) \) denotes a valid range, \( \text{il} \) designates an object of type \( \text{initializer\_list}\langle \text{value\_type} \rangle \), \( n \) denotes a value of type \( X::\text{size\_type} \), \( p \) 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 \( X::\text{value\_type} \), and \( rv \) denotes a non-const rvalue of \( X::\text{value\_type} \). Args denotes a template parameter pack; args denotes a function parameter pack with the pattern \( \text{Args&&} \).

The complexities of the expressions are sequence dependent.

3 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 \( X::\text{allocator\_type} \) if the qualified-id \( X::\text{allocator\_type} \) is valid and denotes a type (13.10.2) and \( \text{allocator}\langle T \rangle \) if it doesn’t, \( i \) and \( j \) denote iterators that meet the \( \text{Cpp17InputIterator} \) requirements and refer to elements implicitly convertible to \( \text{value\_type} \), \( [i, j) \) denotes a valid range, \( \text{il} \) designates an object of type \( \text{initializer\_list}\langle \text{value\_type} \rangle \), \( n \) denotes a value of type \( X::\text{size\_type} \), \( p \) 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 \( X::\text{value\_type} \), and \( rv \) denotes a non-const rvalue of \( X::\text{value\_type} \). Args denotes a template parameter pack; args denotes a function parameter pack with the pattern \( \text{Args&&} \).

4 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>
<th>pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X(n, t) )</td>
<td>( X u(n, t); )</td>
<td>Preconditions: ( T ) is ( \text{Cpp17CopyInsertable} ) into ( X ). Postconditions: distance((\text{begin}()), (\text{end})() ) == ( n ). Effects: Constructs a sequence container with ( n ) copies of ( t ).</td>
<td></td>
</tr>
<tr>
<td>( X(i, j) )</td>
<td>( X u(i, j); )</td>
<td>Preconditions: ( T ) is ( \text{Cpp17EmplaceConstructible} ) into ( X ) from ( *i ). For ( \text{vector} ), if the iterator does not meet the ( \text{Cpp17ForwardIterator} ) requirements (23.3.5.4), ( T ) is also ( \text{Cpp17MoveInsertable} ) into ( X ). Postconditions: distance((\text{begin}()), (\text{end})() ) == distance((i, j) ). Effects: Constructs a sequence container equal to the range ( [i, j) ). Each iterator in the range ( [i, j) ) is dereferenced exactly once.</td>
<td></td>
</tr>
<tr>
<td>( X(\text{il}) )</td>
<td>Equivalent to ( X(\text{il}.\text{begin}()), (\text{il}.\text{end})() )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( a = \text{il} )</td>
<td>( X&amp; )</td>
<td>Preconditions: ( T ) is ( \text{Cpp17CopyInsertable} ) into ( X ) and ( \text{Cpp17CopyAssignable} ). Effects: Assigns the range ( [\text{il}.\text{begin}()), (\text{il}.\text{end})() ) into ( a ). All existing elements of ( a ) are either assigned to or destroyed. Returns: ( *\text{this} ).</td>
<td></td>
</tr>
<tr>
<td>( a.\text{emplace}(p, \text{args}) )</td>
<td>iterator</td>
<td>Preconditions: ( T ) is ( \text{Cpp17EmplaceConstructible} ) into ( X ) from ( \text{args} ). For ( \text{vector} ) and ( \text{deque} ), ( T ) is also ( \text{Cpp17MoveInsertable} ) into ( X ) and ( \text{Cpp17MoveAssignable} ). Effects: Inserts an object of type ( T ) constructed with ( \text{std}::\text{forward}\langle\text{Args}\rangle(\text{args})\ldots ) before ( p ). [Note: ( \text{args} ) may directly or indirectly refer to a value in ( a ). — end note]</td>
<td></td>
</tr>
<tr>
<td>( a.\text{insert}(p, t) )</td>
<td>iterator</td>
<td>Preconditions: ( T ) is ( \text{Cpp17CopyInsertable} ) into ( X ). For ( \text{vector} ) and ( \text{deque} ), ( T ) is also ( \text{Cpp17CopyAssignable} ). Effects: Inserts a copy of ( t ) before ( p ).</td>
<td></td>
</tr>
<tr>
<td>( a.\text{insert}(p, rv) )</td>
<td>iterator</td>
<td>Preconditions: ( T ) is ( \text{Cpp17MoveInsertable} ) into ( X ). For ( \text{vector} ) and ( \text{deque} ), ( T ) is also ( \text{Cpp17MoveAssignable} ). Effects: Inserts a copy of ( rv ) before ( p ).</td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Assertion/note pre-/post-condition</td>
<td></td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-------------------------------------</td>
<td></td>
</tr>
<tr>
<td>a.insert(p, n, t)</td>
<td>iterator</td>
<td>Preconditions: T is Cpp17CopyInsertable into X and Cpp17CopyAssignable. Effects: Inserts n copies of t before p.</td>
<td></td>
</tr>
<tr>
<td>a.insert(p, i, j)</td>
<td>iterator</td>
<td>Preconditions: T is Cpp17EmplaceConstructible into X from *i. For vector and deque, T is also Cpp17MoveInsertable into X, Cpp17MoveConstructible, Cpp17MoveAssignable, and swappable (16.5.3.2). 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>
<td></td>
</tr>
<tr>
<td>a.insert(p, il)</td>
<td>iterator</td>
<td>a.insert(p, il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Preconditions: For vector and deque, T is Cpp17MoveAssignable. Effects: Erases the element pointed to by q.</td>
<td></td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>Preconditions: For vector and deque, T is Cpp17MoveAssignable. Effects: Erases the elements in the range [q1, q2).</td>
<td></td>
</tr>
<tr>
<td>a.clear()</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>
<td></td>
</tr>
<tr>
<td>a.assign(i, j)</td>
<td>void</td>
<td>Preconditions: T is Cpp17EmplaceConstructible into X from *i and assignable from *i. For vector, if the iterator does not meet the forward iterator requirements (23.3.5.4), T is also Cpp17MoveInsertable 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>
<td></td>
</tr>
<tr>
<td>a.assign(il)</td>
<td>void</td>
<td>a.assign(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>a.assign(n, t)</td>
<td>void</td>
<td>Preconditions: T is Cpp17CopyInsertable into X and Cpp17CopyAssignable. 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>
<td></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 \texttt{a.insert(p, n, t)} points to the copy of the first element inserted into \texttt{a}, or \texttt{p} if \( n == 0 \).

The iterator returned from \texttt{a.insert(p, i, j)} points to the copy of the first element inserted into \texttt{a}, or \texttt{p} if \( i == j \).

The iterator returned from \texttt{a.insert(p, il)} points to the copy of the first element inserted into \texttt{a}, or \texttt{p} if \texttt{il} is empty.

The iterator returned from \texttt{a.emplace(p, args)} points to the new element constructed from \texttt{args} into \texttt{a}.

The iterator returned from \texttt{a.erase(q)} points to the element immediately following \texttt{q} prior to the element being erased. If no such element exists, \texttt{a.end()} is returned.

The iterator returned by \texttt{a.erase(q1, q2)} points to the element pointed to by \texttt{q2} prior to any elements being erased. If no such element exists, \texttt{a.end()} is returned.

For every sequence container defined in this Clause and in Clause 21:

\begin{enumerate}
\item If the constructor
\begin{verbatim}
    template<class InputIterator>
    X(InputIterator first, InputIterator last,
       const allocator_type& alloc = allocator_type());
\end{verbatim}
is called with a type \texttt{InputIterator} that does not qualify as an input iterator, then the constructor shall not participate in overload resolution.

\item If the member functions of the forms:
\begin{verbatim}
    template<class InputIterator>
    return-type F(const_iterator p,
                   InputIterator first, InputIterator last); // such as insert
\end{verbatim}
\begin{verbatim}
    template<class InputIterator>
    return-type F(InputIterator first, InputIterator last);    // such as append, assign
\end{verbatim}
\begin{verbatim}
    template<class InputIterator>
    return-type F(const_iterator il, const_iterator i2,
                   InputIterator first, InputIterator last); // such as replace
\end{verbatim}
are called with a type \texttt{InputIterator} that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

\item A deduction guide for a sequence container shall not participate in overload resolution if it has an \texttt{InputIterator} template parameter and a type that does not qualify as an input iterator is deduced for that parameter, or if it has an \texttt{Allocator} template parameter and a type that does not qualify as an allocator is deduced for that parameter.
\end{enumerate}

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.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
Expression & Return type & Operational semantics & Container \\
\hline
\texttt{a.front()} & reference; const\_reference & \texttt{*a.begin()} & basic\_string, array, deque, forward\_list, list, vector \\
\texttt{a.back()} & reference; const\_reference & \{ auto tmp = a.end(); --tmp; \return *tmp; \} & basic\_string, array, deque, list, vector \\
\hline
\end{tabular}
\caption{Optional sequence container operations} \label{tab:container.seq.opt}
\end{table}
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, forward_list, list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17EmplaceConstructible into X from args.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns: a.front().</td>
<td></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, list, vector</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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns: a.back().</td>
<td></td>
</tr>
<tr>
<td>a.push_front(t)</td>
<td>void</td>
<td>Effects: Prepends a copy of t.</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17CopyInsertable into X.</td>
<td></td>
</tr>
<tr>
<td>a.push_front(rv)</td>
<td>void</td>
<td>Effects: Prepends a copy of rv.</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17MoveInsertable into X.</td>
<td></td>
</tr>
<tr>
<td>a.push_back(t)</td>
<td>void</td>
<td>Effects: Appends a copy of t.</td>
<td>basic_string, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17CopyInsertable into X.</td>
<td></td>
</tr>
<tr>
<td>a.push_back(rv)</td>
<td>void</td>
<td>Effects: Appends a copy of rv.</td>
<td>basic_string, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: T is Cpp17MoveInsertable into X.</td>
<td></td>
</tr>
<tr>
<td>a.pop_front()</td>
<td>void</td>
<td>Effects: Destroys the first element.</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: a.empty() is false.</td>
<td></td>
</tr>
<tr>
<td>a.pop_back()</td>
<td>void</td>
<td>Effects: Destroys the last element.</td>
<td>basic_string, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Preconditions: a.empty() is false.</td>
<td></td>
</tr>
<tr>
<td>a[n]</td>
<td>reference; const_reference for constant a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>*(a.begin() + n)</td>
<td>*(a.begin() + n)</td>
<td>basic_string, array, deque, vector</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.

§ 22.2.4.1
Table 79: Container types with compatible nodes  

<table>
<thead>
<tr>
<th>map&lt;K, T, C1, A&gt;</th>
<th>map&lt;K, T, C2, A&gt;</th>
</tr>
</thead>
<tbody>
<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_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>

2 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.

3 Class `node_handle` is for exposition only.

4 If a user-defined specialization of `pair` exists for `pair<const Key, T>` or `pair<Key, T>`, where `Key` is the container’s `key_type` and `T` is the container’s `mapped_type`, the behavior of operations involving node handles is undefined.

```cpp
#include <memory>

template<
    unspecified
>

class node-handle {
public:
    // 22.2.4.1, node-handle
    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(except(x.swap(y))) {
        x.swap(y);
    };
};
```

§ 22.2.4.1
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;
14    Returns: ptr_ != nullptr.
[[nodiscard]] bool empty() const noexcept;
15    Returns: ptr_ == nullptr.

22.2.4.5 Modifiers

[container.node.modifiers]

void swap(node-handle& nh)
    noexcept(ator_traits::propagate_on_container_swap::value ||
        ator_traits::is_always_equal::value);
1    Preconditions: !alloc_, or !nh.alloc_, or ator_traits::propagate_on_container_swap::value
        is true, or alloc_ == nh.alloc_.
2    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

[container.insert.return]

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

[associative.reqmts]

Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of
associative containers: `set`, `multiset`, `map` and `multimap`.

1 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.

2 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: 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.

3 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.

4 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: `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]

7 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: 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]

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 (15.10.2), `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`, `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>
<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 <code>Cpp17Erasable</code> from <code>X</code></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 <code>Cpp17Erasable</code> from <code>X</code></td>
<td>compile time</td>
</tr>
<tr>
<td>X::key_compare</td>
<td>Compare</td>
<td>Preconditions: <code>key_compare</code> is <code>Cpp17CopyConstructible</code>.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_compare</td>
<td>a binary predicate type</td>
<td>is the same as <code>key_compare</code> 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 <code>X</code>.</td>
<td>see §22.2.4</td>
<td>compile time</td>
</tr>
</tbody>
</table>

<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(c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u(c);</td>
<td></td>
<td>Effects: Constructs an empty constant container. Uses a copy of <code>c</code> as a comparison object.</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>X()</td>
<td>X u;</td>
<td>Preconditions: key_compare meets the Cpp17DefaultConstructible requirements. Effects: Constructs an empty container. Uses Compare() as a comparison object.</td>
<td>constant</td>
</tr>
<tr>
<td>X(i,j,c)</td>
<td>X u(i,j,c);</td>
<td>Preconditions: value_type is Cpp17EmplaceConstructible into X from *i. Effects: Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.</td>
<td>N log N in general, where N has the value distance(i, j); linear if [i, j) is sorted with value_comp()</td>
</tr>
<tr>
<td>X(i,j)</td>
<td>X u(i,j);</td>
<td>Preconditions: key_compare meets the Cpp17DefaultConstructible requirements. value_type is Cpp17EmplaceConstructible into X from *i. Effects: Same as above, but uses Compare() as a comparison object.</td>
<td>same as above</td>
</tr>
<tr>
<td>X(il)</td>
<td>same as X(il.begin(), il.end());</td>
<td>same as above</td>
<td></td>
</tr>
<tr>
<td>X(il,c)</td>
<td>same as X(il.begin(), il.end(), c)</td>
<td>same as above</td>
<td></td>
</tr>
<tr>
<td>a = il</td>
<td>X&amp;</td>
<td>Preconditions: value_type 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.</td>
<td>N log N in general, where N has the value il.size() + a.size(); linear if [il.begin(), il.end()] is sorted with value_comp()</td>
</tr>
<tr>
<td>b.key_-comp()</td>
<td>X::key_-compare</td>
<td>Returns: the comparison object out of which b was constructed.</td>
<td>constant</td>
</tr>
<tr>
<td>b.value_-comp()</td>
<td>X::value_-compare</td>
<td>Returns: an object of value_compare constructed out of the comparison object</td>
<td>constant</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.emplace(args)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td><code>Preconditions: value_type is </code>Cpp17EmplaceConstructible into X from args. Effects: Inserts a value_type object t constructed with <code>std::forward&lt;Args&gt;(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. </code>logarithmic`</td>
<td></td>
</tr>
<tr>
<td><code>a_eq.emplace(args)</code></td>
<td><code>iterator</code></td>
<td><code>Preconditions: value_type is </code>Cpp17EmplaceConstructible into X from args. Effects: Inserts a value_type object t constructed with <code>std::forward&lt;Args&gt;(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. </code>logarithmic`</td>
<td></td>
</tr>
<tr>
<td><code>aemplace_hint(p, args)</code></td>
<td><code>iterator</code></td>
<td>equivalent to <code>aemplace(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 element is inserted as close as possible to the position just prior to p. <code>logarithmic in general, but amortized constant if the element is inserted right before p</code></td>
<td></td>
</tr>
<tr>
<td><code>a_uniq.insert(t)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td><code>Preconditions: If t is a non-const rvalue, value_type is </code>Cpp17MoveInsertable into X; otherwise, value_type is <code>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. </code>logarithmic`</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_eq.</td>
<td>iterator</td>
<td>Preconditions: If ( t ) is a non-const rvalue, ( \text{value	ype} ) is \text{Cpp17MoveInsertable} into ( X ); otherwise, ( \text{value	ype} ) is \text{Cpp17CopyInsertable} into ( X ). Effects: Inserts ( t ) 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.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>( \text{insert}(t) )</td>
<td>iterator</td>
<td>Preconditions: If ( t ) is a non-const rvalue, ( \text{value	ype} ) is \text{Cpp17MoveInsertable} into ( X ); otherwise, ( \text{value	ype} ) is \text{Cpp17CopyInsertable} into ( X ). Effects: Inserts ( t ) if and only if there is no element with key equivalent to the key of ( t ) in containers with unique keys; always inserts ( t ) in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of ( t ). ( t ) is inserted as close as possible to the position just prior to ( p ).</td>
<td>logarithmic in general, but amortized constant if ( t ) is inserted right before ( p ).</td>
</tr>
<tr>
<td>a.insert(p, ( \text{iterator} )) ( t )</td>
<td>Preconditions: If ( t ) is a non-const rvalue, ( \text{value	ype} ) is \text{Cpp17MoveInsertable} into ( X ); otherwise, ( \text{value	ype} ) is \text{Cpp17CopyInsertable} into ( X ). Effects: Inserts ( t ) if and only if there is no element with key equivalent to the key of ( t ) in containers with unique keys; always inserts ( t ) in containers with equivalent keys. Always returns the iterator pointing to the element with key equivalent to the key of ( t ). ( t ) is inserted as close as possible to the position just prior to ( p ).</td>
<td>( N \log(a.size() + N) ), where ( N ) has the value ( \text{distance}(i, j) )</td>
<td></td>
</tr>
<tr>
<td>a.insert(i, ( \text{void} )) ( j )</td>
<td>Preconditions: ( \text{value	ype} ) is \text{Cpp17EmplaceConstructible} into ( X ) from ( \ast i ). Neither ( i ) nor ( j ) are iterators into ( a ). Effects: Inserts each element from the range ([i, j)) 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 ( \text{distance}(i, j) )</td>
<td></td>
</tr>
<tr>
<td>a.insert( ( \text{void} )) ( il )</td>
<td>equivalent to ( \text{a.insert(il.begin(), il.end())} )</td>
<td></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>
</table>
| a_uniq.     | insert_.insert_.return_type | **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(). | logarithmic |
| a_eq.       | insert_.iterator | **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. | 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.insert(p, nh)</code></td>
<td>iterator</td>
<td>Preconditions: <code>nh</code> is empty or <code>a.get_allocator()</code> == <code>nh.get_allocator()</code>. Effects: If <code>nh</code> is empty, has no effect and returns <code>a.end()</code>. Otherwise, inserts the element owned by <code>nh</code> 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 <code>nh</code> 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 <code>p</code>. Postconditions: <code>nh</code> is empty if insertion succeeds, unchanged if insertion fails.</td>
<td>logarithmic in general, but amortized constant if the element is inserted right before <code>p</code>.</td>
</tr>
<tr>
<td><code>a.extract(k)</code></td>
<td>node_type</td>
<td>Effects: Removes the first element in the container with key equivalent to <code>k</code>. Returns: A <code>node_type</code> owning the element if found, otherwise an empty <code>node_type</code>.</td>
<td>(\log(a.size()))</td>
</tr>
<tr>
<td><code>a.extract(q)</code></td>
<td>node_type</td>
<td>Effects: Removes the element pointed to by <code>q</code>. Returns: 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>Preconditions: <code>a.get_allocator()</code> == <code>a2.get_allocator()</code>. Effects: Attempts to extract each element in <code>a2</code> and insert it into <code>a</code> using the comparison object of <code>a</code>. In containers with unique keys, if there is an element in <code>a</code> with key equivalent to the key of an element from <code>a2</code>, then that element is not extracted from <code>a2</code>. Postconditions: Pointers and references to the transferred elements of <code>a2</code> refer to those same elements but as members of <code>a</code>. Iterators referring to the transferred elements will continue to refer to their elements, but they now behave as iterators into <code>a</code>, not into <code>a2</code>. Throws: Nothing unless the comparison object throws.</td>
<td>(N \log(a.size()+N),) where (N) has the value <code>a2.size()</code> ()</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>a.erase(k)</td>
<td>size_type</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()) + \text{a.count}(k) )</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</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>a.erase(r)</td>
<td>iterator</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>a.erase(q1, q2)</td>
<td>iterator</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 ( \text{distance}(q1, q2) ).</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Effects: Equivalent to a.erase(a.begin(), a.end()). Postconditions: a.empty() is true.</td>
<td>linear in a.size()</td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator;</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>const_iterator for constant b.</td>
<td></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 ( r ) such that ( \text{!c}(r, \text{ke}) ) &amp;&amp; ( \text{!c}(\text{ke}, r) ), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>const_iterator for constant a_tran</td>
<td></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>( \log(b.size()) + b.count(k) )</td>
</tr>
<tr>
<td>a_tran.count(ke)</td>
<td>size_type</td>
<td>Returns: The number of elements with key ( r ) such that ( \text{!c}(r, \text{ke}) ) &amp;&amp; ( \text{!c}(\text{ke}, r) )</td>
<td>( \log(a_tran.size()) + a_tran.count(ke) )</td>
</tr>
<tr>
<td>b.contains(k)</td>
<td>bool</td>
<td>Effects: Equivalent to: return b.find(k) (!=) b.end();</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran.contains(ke)</td>
<td>bool</td>
<td>Effects: Equivalent to: return a_tran.find(ke) (!=) a_tran.end();</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</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b.lower_bound(k)</code></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. <code>lower_bound(kl)</code></td>
<td>iterator; const_iterator for constant a_tran</td>
<td>Returns: An iterator pointing to the first element with key r such that !c(r, kl), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>b.upper_bound(k)</code></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. <code>upper_bound(ku)</code></td>
<td>iterator; const_iterator for constant a_tran</td>
<td>Returns: An iterator pointing to the first element with key r such that c(ku, r), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>b.equal_range(k)</code></td>
<td><code>pair&lt;iterator, iterator&gt;</code>; <code>pair&lt;const_iterator, const_iterator&gt;</code> 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. <code>equal_range(ke)</code></td>
<td><code>pair&lt;iterator, iterator&gt;</code>; <code>pair&lt;const_iterator, const_iterator&gt;</code> 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:

\[ \text{value_comp}(*j, *i) = false \]

12 For associative containers with unique keys the stronger condition holds:

\[ \text{value_comp}(*i, *j) \neq false \]

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

§ 22.2.6
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.

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 (13.10.2).

A deduction guide for an associative container shall not participate in overload resolution if any of the following are true:

1. It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
2. It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
3. It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

### 22.2.6.1 Exception safety guarantees

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).

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.

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

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`.

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.

Each unordered associative container is parameterized by `Key`, by a function object type `Hash` that meets the Cpp17Hash requirements (16.5.3.4) 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.

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.

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: 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.

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.

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.
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: 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 (13.10.2),
- 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,
- eq denotes a possibly const value of type hasher,
- 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 81: Unordered associative container requirements (in addition to container) [tab:container.hash.req]
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>X::mapped_type (unordered_map and unordered_multimap only)</td>
<td>T</td>
<td>Compile time</td>
<td></td>
</tr>
<tr>
<td>X::value_type (unordered_set and unordered_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 (unordered_map and unordered_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::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>$O(n)$</td>
</tr>
<tr>
<td>X a(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 key_equal() as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

§ 22.2.7
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>( X(n) ) ( X a(n); )</td>
<td>( X )</td>
<td>Preconditions: ( \text{hasher} ) and ( \text{key_equal} ) meet the Cpp17DefaultConstructible requirements. Effects: Constructs an empty container with at least ( n ) buckets, using ( \text{hasher()} ) as the hash function and ( \text{key_equal()} ) as the key equality predicate.</td>
<td>( \mathcal{O}(n) )</td>
</tr>
<tr>
<td>( X() ) ( X a; )</td>
<td>( X )</td>
<td>Preconditions: ( \text{hasher} ) and ( \text{key_equal} ) meet the Cpp17DefaultConstructible requirements. Effects: Constructs an empty container with an unspecified number of buckets, using ( \text{hasher()} ) as the hash function and ( \text{key_equal()} ) as the key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td>( X(i, j, n, hf, eq) ) ( X a(i, j, n, hf, eq); )</td>
<td>( X )</td>
<td>Preconditions: ( \text{value_type} ) is Cpp17EmplaceConstructible into ( X ) from ( *i ). Effects: 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 ( \mathcal{O}(N) ) (( N ) is distance((i, j))), worst case ( \mathcal{O}(N^2) )</td>
</tr>
<tr>
<td>( X(i, j, n, hf) ) ( X a(i, j, n, hf); )</td>
<td>( X )</td>
<td>Preconditions: ( \text{key_equal} ) meets the Cpp17DefaultConstructible requirements. ( \text{value_type} ) is Cpp17EmplaceConstructible into ( X ) from ( *i ). Effects: Constructs an empty container with at least ( n ) buckets, using ( hf ) as the hash function and ( \text{key_equal()} ) as the key equality predicate, and inserts elements from ( [i, j) ) into it.</td>
<td>Average case ( \mathcal{O}(N) ) (( N ) is distance((i, j))), worst case ( \mathcal{O}(N^2) )</td>
</tr>
<tr>
<td>( X(i, j, n) ) ( X a(i, j, n); )</td>
<td>( X )</td>
<td>Preconditions: ( \text{hasher} ) and ( \text{key_equal} ) meet the Cpp17DefaultConstructible requirements. ( \text{value_type} ) is Cpp17EmplaceConstructible into ( X ) from ( *i ). Effects: Constructs an empty container with at least ( n ) buckets, using ( \text{hasher()} ) as the hash function and ( \text{key_equal()} ) as the key equality predicate, and inserts elements from ( [i, j) ) into it.</td>
<td>Average case ( \mathcal{O}(N) ) (( N ) is distance((i, j))), worst case ( \mathcal{O}(N^2) )</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>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>Average case (\Theta(N) ) ((N) is distance(i, j)), worst case (\Theta(N^2))</td>
</tr>
<tr>
<td>X a(i, j);</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(il)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end()).</td>
<td>Same as X(il.begin(), il.end()).</td>
</tr>
<tr>
<td>X(il, n)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end(), n).</td>
<td>Same as X(il.begin(), il.end(), n).</td>
</tr>
<tr>
<td>X(il, n, hf)</td>
<td>X</td>
<td>Same as X(il.begin(), il.end(), n, hf).</td>
<td>Same as X(il.begin(), il.end(), n, hf).</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>Same as X(il.begin(), il.end(), n, hf, eq).</td>
</tr>
<tr>
<td>X(b)</td>
<td>X</td>
<td>Copy constructor. In addition to the requirements of Table 73, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
</tr>
<tr>
<td>X a(b);</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = b</td>
<td>X&amp;</td>
<td>Copy assignment operator. In addition to the requirements of Table 73, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
</tr>
<tr>
<td>a = il</td>
<td>X&amp;</td>
<td><strong>Preconditions:</strong> value_type is Cpp17CopyInsertable into X and Cpp17CopyAssignable. <strong>Effects:</strong> Assigns the range [il.begin(), il.end()) into a. All existing elements of a are either assigned to or destroyed.</td>
<td>Same as a = X(il).</td>
</tr>
<tr>
<td>b.hash_function()</td>
<td>hasher</td>
<td><strong>Returns:</strong> b’s hash function.</td>
<td>constant</td>
</tr>
<tr>
<td>b.key_eq()</td>
<td>key_equal</td>
<td><strong>Returns:</strong> b’s key equality predicate.</td>
<td>constant</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><code>a_uniq.emplace(args)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td>Preconditions: value_type is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Inserts a value_type 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 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.</td>
<td>Average case ( O(1) ), worst case ( O(a_{uniq}.size()) ).</td>
</tr>
<tr>
<td><code>a_eq.emplace(args)</code></td>
<td><code>iterator</code></td>
<td>Preconditions: value_type is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Inserts a value_type 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><code>a.emplace_hint(p, args)</code></td>
<td><code>iterator</code></td>
<td>Preconditions: value_type 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 const_iterator p 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><code>a_uniq.insert(t)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td>Preconditions: If t is a non-const rvalue, value_type is <code>Cpp17MoveInsertable</code> into <code>X</code>; otherwise, value_type is <code>Cpp17CopyInsertable</code> into <code>X</code>. 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>Average case ( O(1) ), worst case ( O(a_{uniq}.size()) ).</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_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 an iterator pointing to the newly inserted element.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a_{eq}.size())$.</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> Equivalent to <code>a.insert(t)</code>. Return value is an iterator pointing to the element with the key equivalent to that of <code>t</code>. The iterator <code>p</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</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> Equivalent to <code>a.insert(t)</code> for each element in <code>[i,j)</code>.</td>
<td>Average case $\mathcal{O}(N)$, where <code>N</code> is <code>distance(i, j)</code>, worst case $\mathcal{O}(N(a.size() + 1))$.</td>
</tr>
<tr>
<td><code>a.insert(iil)</code></td>
<td>void</td>
<td>Same as <code>a.insert(iil.begin(), iil.end())</code>.</td>
<td>Same as <code>a.insert(iil.begin(), iil.end())</code>.</td>
</tr>
<tr>
<td><code>a_uniq.insert(nh)</code></td>
<td>insert_return_type</td>
<td><em>Preconditions:</em> <code>nh</code> is empty or <code>a_uniq.get_allocator() == nh.get_allocator()</code>. <em>Effects:</em> If <code>nh</code> is empty, has no effect. Otherwise, inserts the element owned by <code>nh</code> if and only if there is no element in the container with a key equivalent to <code>nh.key()</code>. <em>Postconditions:</em> If <code>nh</code> 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 <code>nh</code>, and <code>position</code> points to an element with a key equivalent to <code>nh.key()</code>.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a_{uniq}.size())$.</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</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_eq.insert(nh)</code></td>
<td>iterator</td>
<td><strong>Preconditions:</strong> nh is empty or a_eq.get_allocator() == nh.get_allocator(). <strong>Effects:</strong> 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. <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>iterator</td>
<td><strong>Preconditions:</strong> nh is empty or a.get_allocator() == nh.get_allocator(). <strong>Effects:</strong> If nh is empty, has no effect and returns a.end(). Otherwise, inserts the element owned by nh if and only if there is no element with key equivalent to nh.key() 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 nh.key(). 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>
<tr>
<td><code>a.extract(k)</code></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 $O(1)$, worst case $O(a.size())$.</td>
</tr>
<tr>
<td><code>a.extract(q)</code></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 $O(1)$, 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</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.merge(a2)</td>
<td>void</td>
<td>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.</td>
<td>Average case $\mathcal{O}(N)$, where $N$ is a2.size(), worst case $\mathcal{O}(N \cdot a.size() + N)$.</td>
</tr>
<tr>
<td>a.erase(k)</td>
<td>size_type</td>
<td>Effects: Erases all elements with key equivalent to k. Returns: The number of elements erased.</td>
<td>Average case $\mathcal{O}(a.count(k))$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Effects: Erases the element pointed to by q. Returns: The iterator immediately following q prior to the erasure.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(r)</td>
<td>iterator</td>
<td>Effects: Erases the element pointed to by r. Returns: The iterator immediately following r prior to the erasure.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>Effects: Erases all elements in the range [q1, q2). Returns: The iterator immediately following the erased elements prior to the erasure.</td>
<td>Average case linear in $\mathcal{O}(\text{distance}(q1, q2))$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Effects: Erases all elements in the container. Postconditions: a.empty() is true</td>
<td>Linear in $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator; const_iterator for const b.</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 $\mathcal{O}(1)$, worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td>a_tran.find(ke)</td>
<td>iterator; const_iterator for const a_tran.</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 $\mathcal{O}(1)$, worst case $\mathcal{O}(a_tran.size())$.</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>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 might ever contain.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.bucket(k)</td>
<td>size_type</td>
<td>Preconditions: $b.bucket_count() &gt; 0$. Returns: The index of the bucket in which elements with keys equivalent to k would be found, if any such element existed. Postconditions: The return value shall be in the range $[0, b.bucket_count())$.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.bucket_size(n)</td>
<td>size_type</td>
<td>Preconditions: n shall be in the range $[0, b.bucket_count())$. Returns: The number of elements in the n\textsuperscript{th} bucket.</td>
<td>$\Theta(b.bucket_size(n))$</td>
</tr>
<tr>
<td>b.begin(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td>Preconditions: n is in the range $[0, b.bucket_count())$. Returns: An iterator referring to the first element in the bucket. If the bucket is empty, then b.begin(n) == b.end(n).</td>
<td>Constant</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>b.end(n)</td>
<td>local_iterator;</td>
<td>Preconditions: n is in the range</td>
<td>Constant</td>
</tr>
<tr>
<td>const_local_iterator</td>
<td>for const</td>
<td>[0, b.bucket_count()). Returns: An</td>
<td></td>
</tr>
<tr>
<td>b.</td>
<td>iterator</td>
<td>iterator which is the past-the-end</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>value for the bucket.</td>
<td></td>
</tr>
<tr>
<td>b.cbegin(n)</td>
<td>const_local_iterator</td>
<td>Preconditions: n shall be in the</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>range [0, b.bucket_count()). Returns</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>An iterator referring to the first</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>element in the bucket. If the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bucket is empty, then</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>b.cbegin(n) == b.cend(n).</td>
<td></td>
</tr>
<tr>
<td>b.cend(n)</td>
<td>const_local_iterator</td>
<td>Preconditions: n is in the range</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0, b.bucket_count()). Returns:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>An iterator which is the past-the-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>end value for the bucket.</td>
<td></td>
</tr>
<tr>
<td>b.load_factor()</td>
<td>float</td>
<td>Returns: The average number of</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elements per bucket.</td>
<td></td>
</tr>
<tr>
<td>b.max_load_factor()</td>
<td>float</td>
<td>Returns: A positive number that</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the container attempts to keep the</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>load factor less than or equal to.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The container automatically</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>increases the number of buckets</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>as necessary to keep the load</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>factor below this number.</td>
<td></td>
</tr>
<tr>
<td>a.max_load_factor(z)</td>
<td>void</td>
<td>Preconditions: z is positive.</td>
<td>Constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>May change the container’s</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>maximum load factor, using z as a</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>hint.</td>
<td></td>
</tr>
<tr>
<td>a.rehash(n)</td>
<td>void</td>
<td>Postconditions:</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.bucket_count() &gt;=</td>
<td>linear in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.size() /</td>
<td>a.size(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.max_load_factor() and</td>
<td>worst case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.bucket_count() &gt;= n.</td>
<td>quadratic.</td>
</tr>
<tr>
<td>a.reserve(n)</td>
<td>void</td>
<td>Same as a.rehash(ceil(n /</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.max_load_factor())).</td>
<td>linear in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a.size(),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>worst case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>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 \( O(N^2) \), e.g., for a pathologically bad hash function). The behavior of a program that uses operator== or operator!= on unordered containers is undefined unless

§ 22.2.7 817
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 (13.10.2).

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.

22.2.7.1 Exception safety guarantees [unord.req.except]

1 For unordered associative containers, no clear() function throws an exception. erase(k) does not throw an exception unless that exception is thrown by the container’s Hash or Pred object (if any).

2 For unordered associative containers, if an exception is thrown by any operation other than the container’s hash function from within an insert or emplace function inserting a single element, the insertion has no effect.

3 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).

4 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 [sequences]

22.3.1 In general [sequences.general]

1 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.

2 The following exposition-only alias template may appear in deduction guides for sequence containers:

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.1
```

224) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.

§ 22.3.2 818
namespace std {
    // 22.3.7, class template 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);

    // 22.3.7.4, specialized algorithms
    template<class T, size_t N>
    constexpr void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

    // 22.3.7.6, array creation functions
    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]);

    // 22.3.7.7, tuple interface
    template<class T> struct tuple_size;
    template<size_t I, class T> struct tuple_element;
    template<class T, size_t N>
    struct tuple_size<array<T, N>>;
    template<size_t I, class T, size_t N>
    struct tuple_element<I, array<T, 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;
}

22.3.3 Header <deque> synopsis

#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.1

namespace std {
    // 22.3.8, class template deqe
    template<class T, class Allocator = allocator<T>> class deqe;

    template<class T, class Allocator>
    bool operator==(const deqe<T, Allocator>& x, const deqe<T, Allocator>& y);

    template<class T, class Allocator>
    synth-three-way-result<T> operator<>=(const deqe<T, Allocator>& x,
                                       const deqe<T, Allocator>& y);

    template<class T, class Allocator>
    void swap(deqe<T, Allocator>& x, deqe<T, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

    template<class T, class Allocator, class U>
    typename deqe<T, Allocator>::size_type
    erase(deqe<T, Allocator>& x, deqe<T, Allocator>& y, size_t c,
          const U& value);

    template<class T, class Allocator, class Predicate>
    typename deqe<T, Allocator>::size_type
    erase_if(deqe<T, Allocator>& x, Predicate pred);

§ 22.3.3
namespace pmr {
    template<class T>
    using deque = std::deque<T, polymorphic_allocator<T>>;
}

22.3.4 Header <forward_list> synopsis

#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.1

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

#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.1

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)));

    template<class T, class Allocator, class U>
    typename list<T, Allocator>::size_type
        erase(list<T, Allocator>& c, const U& value);

    template<class T, class Allocator, class Predicate>
    typename list<T, Allocator>::size_type
        erase_if(list<T, Allocator>& c, Predicate pred);
}

§ 22.3.5
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 17.10.1
```

namespace std {
```
// 22.3.11, class template vector
template<class T, class Allocator = allocator<T>> class vector;
```

```cpp
template<class T, class Allocator>
constexpr bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
```

```cpp
template<class T, class Allocator>
constexpr synth-three-way-result<T> operator<=>(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
```

```cpp
template<class T, class Allocator>
constexpr void swap(vector<T, Allocator>& x, vector<T, Allocator>& y) noexcept(noexcept(x.swap(y)));
```

```cpp
template<class T, class Allocator, class U>
constexpr typename vector<T, Allocator>::size_type erase(vector<T, Allocator>& c, const U& value);
```

```cpp
template<class T, class Allocator, class Predicate>
constexpr typename vector<T, Allocator>::size_type erase_if(vector<T, Allocator>& c, Predicate pred);
```

```
// 22.3.12, class vector<bool>
template<class Allocator> class vector<bool, Allocator>;
```

```
// hash support
```

```cpp
template<class T> struct hash;
```

```cpp
template<class Allocator> struct hash<vector<bool, Allocator>>;
```

namespace pmr {
```
template<class T>
using vector = std::vector<T, polymorphic_allocator<T>>;
```}

22.3.7 Class template array

22.3.7.1 Overview

1 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.

2 An `array` is an aggregate (9.4.1) that can be list-initialized with up to N elements whose types are convertible to T.

3 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.

4 `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.

5 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;

        // 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<class T, class... U>
    array(T, U...) -> array<T, 1 + sizeof...(U)>;
}

22.3.7.2 Constructors, copy, and assignment

1 The conditions for an aggregate (9.4.1) shall be met. Class array relies on the implicitly-declared special member functions (11.4.4.1, 11.4.6, and 11.4.4.2) 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.

```cpp
template<class T, class... U>
array(T, U...) -> array<T, 1 + sizeof...(U)>;
```

*Mandates: *(is_same_v<T, U> && ...) is true.*

### 22.3.7.3 Member functions [array.members]

**constexpr size_type size() const noexcept;**

*Returns:* `N`.  

```cpp
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
```

*Returns: A pointer such that [data(), data() + size()) is a valid range. For a non-empty array, data() == addressof(front()).*  

```cpp
constexpr void fill(const T& u);
```

*Effects: As if by `fill_n(begin(), N, u)`.*  

```cpp
constexpr void swap(array& y) noexcept(is_nothrow_swappable_v<T>);
```

*Effects: Equivalent to `swap_ranges(begin(), end(), y.begin())`.*  

[Note: Unlike the `swap` function for other containers, `array::swap` takes linear time, may 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]

```cpp
template<class T, size_t N>
constexpr void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));
```

*Constraints: * `N == 0` or `is_swappable_v<T>` is true.  

*Effects: As if by `x.swap(y)`.*  

*Complexity: Linear in `N`.*

### 22.3.7.5 Zero-sized arrays [array.zero]

`array` shall provide support for the special case `N == 0`.  

In the case that `N == 0`, `begin() == end() == unique value`. The return value of `data()` is unspecified.  

The effect of calling `front()` or `back()` for a zero-sized array is undefined.  

Member function `swap()` shall have a non-throwing exception specification.

### 22.3.7.6 Array creation functions [array.creation]

```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]) }}`.  

§ 22.3.7.6
22.3.7.7 Tuple interface

```
template<class T, size_t N>
struct tuple_size<array<T, N>> : integral_constant<size_t, N> { }

template<size_t I, class T, size_t N>
struct tuple_element<I, array<T, N>> {
  using type = T;
};
```

**Mandates:** \( I < N \) is true.

```
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;
template<size_t I, class T, size_t N>
constexpr const T& get(const array<T, N>& a) noexcept;
template<size_t I, class T, size_t N>
constexpr const T&& get(const 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 (23.3.5.6). 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.

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class deque {
  public:
    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>;

    // 22.3.8.2, construct/copy/destroy
    deque() : deque(Allocator()) { }
    explicit deque(const Allocator&) { }
    explicit deque(size_type n, const Allocator&) = Allocator();
    deque(size_type n, const T& value, const Allocator&) = Allocator();
    template<class InputIterator>
    deque(InputIterator first, InputIterator last, const Allocator&) = Allocator();
    deque(const deque&) = Allocator();
    deque(deque&&) = Allocator();
    deque(const deque&, const Allocator&);
    deque(deque&&, const Allocator&);
  }
```
deque(initializer_list<T>, const Allocator& = Allocator());

~deque();
deque& operator=(const deque& x);
deque& operator=(deque&& x) noexcept(algorithm_traits<Allocator>::is_always_equal::value);
deque& 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;

// 22.3.8.3, capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;
void resize(size_type sz);
void resize(size_type sz, const T& c);
void shrink_to_fit();

// 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;
};

template<class InputIterator, class Allocator = allocator<iter-value-type<InputIterator>>>
deque(InputIterator, InputIterator, Allocator = Allocator())
-> deque<iter-value-type<InputIterator>, Allocator>;

// swap
template<class T, class Allocator>
void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
  noexcept(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());
  Effects: Constructs a deque with n default-inserted elements using the specified allocator.
  Preconditions: T is Cpp17DefaultInsertable into *this.
  Complexity: Linear in n.

deque(size_type n, const T& value, const Allocator& = Allocator());
  Effects: Constructs a deque with n copies of value, using the specified allocator.
  Preconditions: T is Cpp17CopyInsertable into *this.
  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 [deque.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: The request is non-binding to allow latitude for implementation-specific

§ 22.3.8.3 826
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);**

1. **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.

2. **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.

3. **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.

**iterator erase(const_iterator position);**

**iterator erase(const_iterator first, const_iterator last);**

**void pop_front();**

**void pop_back();**

4. **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: pop_front and pop_back are erase operations. —end note]

5. **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.

6. **Throws:** Nothing unless an exception is thrown by the assignment operator of T.

### 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);**

1. **Effects:** Equivalent to:

   ```cpp
goto it = remove(c.begin(), c.end(), value);
goto r = distance(it, c.end());
c.erase(it, c.end());
```
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: 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: 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, ranges that are modified, such as those supplied to `erase` and `splice`, must be open at the beginning. — end note]
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 (16.5.3.5.1). T shall be complete before any member of the resulting specialization of forward_list is referenced.

22.3.9.2 Constructors, copy, and assignment

explicit forward_list(const Allocator&);
1  Effects: Constructs an empty forward_list object using the specified allocator.
2  Complexity: Constant.

explicit forward_list(size_type n, const Allocator& = Allocator());
3  Preconditions: T is Cpp17DefaultInsertable into *this.
4  Effects: Constructs a forward_list object with n default-inserted elements using the specified allocator.
5  Complexity: Linear in n.

forward_list(size_type n, const T& value, const Allocator& = Allocator());
6  Preconditions: T is Cpp17CopyInsertable into *this.
7  Effects: Constructs a forward_list object with n copies of value using the specified allocator.
8  Complexity: Linear in n.

template<class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
9  Effects: Constructs a forward_list object equal to the range [first, last).
10  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;
1  Returns: A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by begin().
2  Effects: cbefore_begin() is equivalent to const_cast<forward_list const&>(*this).before_begin().
3  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.

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.

void push_front(const T& x);
void push_front(T&& x);

Effects: Inserts a copy of x at the beginning of the list.

void pop_front();

Effects: As if by erase_after(before_begin()).

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.

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.

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.

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.

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.
iterator erase_after(const_iterator position);

19 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.

iterator erase_after(const_iterator position, const_iterator last);

23 Preconditions: All iterators in the range (position, last) are dereferenceable.
   Effects: Erases the elements in the range (position, last).
   Returns: last.
   Throws: Nothing.

void resize(size_type sz);  
27 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.

void resize(size_type sz, const value_type& c);  
29 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.

void clear() noexcept;
31 Effects: Erases all elements in the range [begin(), end()).
   Remarks: Does not invalidate past-the-end iterators.

§ 22.3.9.6  Operations [forwardlist.ops] 832

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.

void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);
2 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: Θ(distance(x.begin(), x.end()))

void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);
6 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: $O(1)$

```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: $O(distance(first, last))$

```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.

Remarks: Stable (16.5.5.8).

Complexity: Exactly `distance(begin(), end())` applications of the corresponding predicate.

```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`.

Remarks: Stable (16.5.5.8).

Complexity: At most `distance(begin(), end()) + distance(x.begin(), x.end()) - 1` comparisons.
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.

Remarks: Stable (16.5.5.8).

Complexity: Approximately $N \log N$ comparisons, where $N$ is distance(begin(), end()).

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

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, [ camper l] (auto& elem) { return elem == value;});

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.\footnote{These member functions are only provided by containers whose iterators are random access iterators.} 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 = std::reverse_iterator<iterator>
      using const_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;
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, 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;
};

/* 3 An incomplete type T may be used when instantiating list if the allocator meets the allocator completeness requirements (16.5.3.5.1). T shall be complete before any member of the resulting specialization of list is referenced. */

22.3.10.2 Constructors, copy, and assignment [list.cons]

explicit list(const Allocator&);

Effects: Constructs an empty list, using the specified allocator.
Complexity: Constant.

explicit list(size_type n, const Allocator& = Allocator());

Preconditions: T is Cpp17DefaultInsertable into *this.
Effects: Constructs a list with n default-inserted elements using the specified allocator.
Complexity: Linear in n.

list(size_type n, const T& value, const Allocator& = Allocator());

Preconditions: T is Cpp17CopyInsertable into *this.
Effects: Constructs a list with n copies of value, using the specified allocator.
Complexity: Linear in $n$.

```cpp
template<class InputIterator>
list(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

Effects: Constructs a list equal to the range [first, last).

Complexity: Linear in distance(first, last).

### 22.3.10.3 Capacity

```cpp
void resize(size_type sz);
```

Preconditions: T is Cpp17DefaultInsertable into *this.

Effects: If size() < sz, appends sz - size() default-inserted elements to the sequence. If sz <= size(), equivalent to:

```cpp
list<T>::iterator it = begin();
advance(it, sz);
erase(it, end());
```

```cpp
void resize(size_type sz, const T& c);
```

Preconditions: T is Cpp17CopyInsertable into *this.

Effects: As if by:

```cpp
if (sz > size())
    insert(end(), sz-size(), c);
else if (sz < size()) {
    iterator i = begin();
    advance(i, sz);
    erase(i, end());
}
else
    // do nothing
```

### 22.3.10.4 Modifiers

```cpp
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);
```

Remarks: Does not affect the validity of iterators and references. If an exception is thrown there are no effects.

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.

```cpp
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 [list.ops]

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 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 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.

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 if addressof(x) == this; otherwise, linear time.

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.

226) As specified in 16.5.3.5, the requirements in this Clause apply only to lists whose allocators compare equal.
Returns: The number of elements erased.

Throws: Nothing unless an exception is thrown by \(*i == value\) or \(\text{pred}(\ast i) != \text{false}\).

Remarks: Stable (16.5.5.8).

Complexity: Exactly \(\text{size()}\) applications of the corresponding predicate.

```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 \([\text{first} + 1, \text{last})\) for which \(*i == *(i-1)\) (for the version of \(\text{unique}\)
with no arguments) or \(\text{pred}(\ast i, \ast(i - 1))\) (for the version of \(\text{unique}\) 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 \(\text{pred}(\ast i, \ast(i - 1))\).

Complexity: If the range \([\text{first}, \text{last})\) is not empty, exactly \((\text{last} - \text{first}) - 1\) 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
\(\text{operator}<\) (for the first two overloads) or \(\text{comp}\) (for the last two overloads), and \(\text{get_allocator()} == x.\text{get_allocator()}\) is \text{true}.

Effects: If \(\text{addressof}(x) == \text{this}\), does nothing; otherwise, merges the two sorted ranges \([\text{begin()}, \text{end()}\) and \([x.\text{begin()}, x.\text{end()}\). The result is a range in which the elements will be sorted in
non-decreasing order according to the ordering defined by \(\text{comp}\); that is, for every iterator \(i\), in the
range other than the first, the condition \(\text{comp}(\ast i, \ast(i - 1))\) will be \text{false}. Pointers and references to
the moved elements of \(x\) now refer to those same elements but as members of \(*\text{this}\). Iterators referring
to the moved elements will continue to refer to their elements, but they now behave as iterators into
\(*\text{this}, \text{not into } x\).

Remarks: Stable (16.5.5.8). If \(\text{addressof}(x) != \text{this}\), the range \([x.\text{begin()}, x.\text{end()}\) is empty
after the merge. No elements are copied by this operation.

Complexity: At most \(\text{size()} + x.\text{size()} - 1\) applications of \(\text{comp}\) if \(\text{addressof}(x) != \text{this}\); other-
wise, no applications of \(\text{comp}\) are performed. If an exception is thrown other than by a comparison
there are no effects.

```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.

```cpp
void sort();
template<class Compare> void sort(Compare comp);
```

Effects: Sorts the list according to the \(\text{operator}<\) or a \(\text{Compare}\) function object. If an exception is
thrown, the order of the elements in \(*\text{this}\) is unspecified. Does not affect the validity of iterators and
references.

Remarks: Stable (16.5.5.8).

Complexity: Approximately \(N \log N\) comparisons, where \(N == \text{size()}\).

### 22.3.10.6 Erasure

```cpp
template<class T, class Allocator, class U>
typename list<T, Allocator>::size_type
erase(list<T, Allocator>& c, const U& value);
```

Effects: Equivalent to:

```cpp
return erase_if(c, [&] { 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 [vector]
22.3.11.1 Overview [vector.overview]

1 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.

2 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.

3 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
      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(size_type n, const T& value, const Allocator& = Allocator());
      template<class InputIterator>
      constexpr vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
      constexpr vector(const vector& x);
      constexpr vector(vector&&) noexcept;
      constexpr vector(const vector&, const Allocator&);
      constexpr vector(vector&&, const Allocator&);
      template<class InputIterator>
      constexpr void assign(InputIterator first, InputIterator last);
      constexpr void assign(size_type n, const T& u);
      constexpr void assign(initializer_list<T>);
      constexpr allocator_type get_allocator() const noexcept;
  } // namespace std

§ 22.3.11.1
// 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;
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& c);
constexpr void reserve(size_type n);
constexpr void shrink_to_fit();

// 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& c);
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;

// data access
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

// modifiers
template<class... Args> 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 reference 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 InputIterator>
constexpr iterator insert(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&);
constexpr void clear() noexcept;

};

template<class InputIterator, class Allocator = allocator<iter_value_type<InputIterator>>> vector(InputIterator, InputIterator, Allocator = Allocator())
-> vector<iter_value_type<InputIterator>>, Allocator>;

§ 22.3.11.1
An incomplete type `T` may be used when instantiating `vector` if the allocator meets the allocator completeness requirements (16.5.3.5.1). `T` shall be complete before any member of the resulting specialization of `vector` is referenced.

### 22.3.11.2 Constructors, copy, and assignment

```cpp
constexpr explicit vector(const Allocator&) noexcept;  // Effects: Constructs an empty vector, using the specified allocator.
```

```cpp
constexpr explicit vector(size_type n, const Allocator& = Allocator());  // Preconditions: T is `Cpp17DefaultInsertable` into `*this`.
```

```cpp
constexpr vector(size_type n, const T& value, const Allocator& = Allocator());  // Preconditions: T is `Cpp17CopyInsertable` into `*this`.
```

```cpp
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.
```

### 22.3.11.3 Capacity

```cpp
constexpr size_type capacity() const noexcept;  // Returns: The total number of elements that the vector can hold without requiring reallocation.
```

```cpp
constexpr void reserve(size_type n);  // Preconditions: T is `Cpp17MoveInsertable` into `*this`.
```

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: If no reallocation happens, they remain valid.]

227) `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.
constexpr void shrink_to_fit();

Preconditions: T is Cpp17MoveInsertable into *this.

Effects: shrink_to_fit is a non-binding request to reduce capacity() to size(). [Note: 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. If an exception is thrown other than by the move constructor of a non-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: If no reallocation happens, they remain valid. — end note]

constexpr void swap(vector& x) noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value);

Effects: Exchanges the contents and capacity() of *this with that of x.

Complexity: Constant time.

constexpr 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.

Remarks: If an exception is thrown other than by the move constructor of a non-Cpp17CopyInsertable T there are no effects.

constexpr 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.

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;

Returns: A pointer such that [data(), data() + size()) is a valid range. For a non-empty vector, data() == addressof(front()).

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);

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.

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

castexpr iterator erase(const_iterator position);
castexpr iterator erase(const_iterator first, const_iterator last);
castexpr void pop_back();

3

Effects: Invalidates iterators and references at or after the point of the erase.

4

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.

5

Throws: Nothing unless an exception is thrown by the assignment operator or move assignment operator
of \( T \).

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);
```

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>
    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>

1

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;
public:
    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(size_type n, const bool& 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());
template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last);
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;
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;

// 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;

§ 22.4 Associative containers

22.4.1 In general

1 The header <map> defines the class templates map and multimap; the header <set> defines the class templates set and multiset.

2 The following exposition-only alias templates may appear in deduction guides for associative containers:

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

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 (20.10.9.2) 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.

4 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:

bool b = x;
x = y;
y = b;

template<class Allocator> struct hash<vector<bool, Allocator>>;

7 The specialization is enabled (20.14.18).
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

#include <compare>              // see 17.11.1
#include <initializer_list>     // see 17.10.1

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);

    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)));

    template<class Key, class T, class Compare, class Allocator, class Predicate>
    typename multimap<Key, T, Compare, Allocator>::size_type
        erase_if(multimap<Key, T, Compare, Allocator>& c, Predicate pred);

    namespace pmr {
        template<class Key, class T, class Compare = less<Key>>
            using map = std::map<Key, T, Compare,
                                polymorphic_allocator<pair<const Key, T>>;
template<class Key, class T, class Compare = less<Key>>
using multimap = std::multimap<Key, T, Compare,
    polymorphic_allocator<pair<const Key, T>>>;
}

22.4.3 Header <set> synopsis

#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.1

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);  
    // 22.4.7, 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>
typename 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>>;
    }
}
22.4.4 Class template map

22.4.4.1 Overview

A `map` is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type `T` based on the keys. The `map` class supports bidirectional iterators.

A `map` 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 `map` also provides most operations described in 22.2.6 for unique keys. This means that a `map` supports the `a_uniq` operations in 22.2.6 but not the `a_eq` operations. For a `map<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on `map` that are not described in one of those tables or for operations where there is additional semantic information.

```cpp
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>;
    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(const 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());

} // namespace std
```
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);
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>);

void clear() noexcept;

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);

};
template<class InputIterator, class Compare = less<
      iter-key-type<InputIterator>>, class Allocator = allocator<
      iter-to-alloc-type<InputIterator>>, 
      iter-mapped-type<InputIterator>>, Compare, Allocator>>
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

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 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<P&<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))`.

```cpp
template<class... Args>
pair<iterator, bool> try_emplace(const key_type& k, 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);
```

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 map from `piecewise_construct, forward_as_tuple(k), forward_as_tuple(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 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 `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 map<Key, T, Compare, Allocator>::size_type
  erase_if(map<Key, T, Compare, Allocator>& c, Predicate pred);
```

Effects: Equivalent to:

1. auto original_size = c.size();
2. for (auto i = c.begin(), last = c.end(); i != last; ) {
3.  if (pred(*i)) {
4.    i = c.erase(i);
5.  } else {
6.    ++i;
7.  }
8.  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.

2. 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.

```cpp
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 = 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;
        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(const 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(algorithm_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;
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(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multimap&) noexcept;
   noexcept(algorithm_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);
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 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 [multimap.cons]

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 [first, last) is already sorted using comp and otherwise
  N log N, where N is last - first.

22.4.5.3 Modifiers [multimap.modifiers]

template<class P> iterator insert(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)).

22.4.5.4 Erasure [multimap.erasure]

template<
class Key, class T, class Compare, class Allocator, class Predicate>
typename multimap<Key, T, Compare, Allocator>::size_type
erase_if(multimap<Key, T, Compare, Allocator>& 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();
22.4.6 Class template set

22.4.6.1 Overview

A 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 set class supports bidirectional iterators.

A 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 set also provides most operations described in 22.2.6 for unique keys. This means that a set supports the a_uniq operations in 22.2.6 but not the a_eq operations. For a set<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on set 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 set {
        public:
            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& = 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(allocator_traits<Allocator>::is_always_equal::value &&
                           is_nothrow_moveAssignable_v<Compare>);
            set& operator=(initializer_list<value_type>);   // allocate
            allocator_type get_allocator() const noexcept;

            // iterators
            iterator begin() noexcept;

            

§ 22.4.6.1 858
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;
22.4.6.2 Constructors, copy, and assignment

```cpp
explicit set(const Compare& comp, const Allocator& = Allocator());
```

**Effects:** Constructs an empty `set` using the specified comparison objects and allocator.

**Complexity:** Constant.

```cpp
template<class InputIterator>
set(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());
```

**Effects:** Constructs an empty `set` using the specified comparison object and allocator, and inserts elements from the range `[first, last)`. 

§ 22.4.2
4 Complexity: Linear in $N$ if the range $[\text{first}, \text{last})$ is already sorted using $\text{comp}$ and otherwise $N \log N$, where $N$ is $\text{last} - \text{first}$.

22.4.6.3 Erasure

```cpp
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);
```

Effects: Equivalent to:

```cpp
class begin() { return (auto i = c.begin(); i != c.end(); ++i) { 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

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.

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.

```cpp
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(multiset&&, const Allocator&);
multiset(initializer_list<value_type>, const Compare& = Compare(),
    const Allocator& = Allocator());

// 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

// iterators

// capacity

// modifiers

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;

// constructors

§ 22.4.7.1 863
22.4.7.2 Constructors

```cpp
explicit multiset(const Compare& comp, const Allocator& = Allocator());
```

1. Effects: Constructs an empty `multiset` using the specified comparison object and allocator.
2. Complexity: Constant.

```cpp
template<class InputIterator>
multiset(InputIterator first, InputIterator last,
const Compare& comp = Compare(), const Allocator& = Allocator());
```

3. Effects: Constructs an empty `multiset` 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`.

22.4.7.3 Erasure

```cpp
template<class Key, class Compare, class Allocator, class Predicate>
type_name multiset<Key, Compare, Allocator> :: size_type
erase_if(multiset<Key, Compare, Allocator>& c, Predicate pred);
```

1. 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 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 17.10.1

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;
} // std
```
// 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>,  
        class Alloc = allocator<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
[unord.set.syn]
#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.1

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)));

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>>;

  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

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.

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>.

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());

  unordered_map(const unordered_map&);
  unordered_map(unordered_map&&);
  explicit unordered_map(const Allocator&);
  unordered_map(const unordered_map&, const Allocator&);
  unordered_map(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_type& a)
    : unordered_map(n, hasher(), key_equal(), a) { }

  unordered_map(size_type n, const key_equal& kf, const allocator_type& a)
    : unordered_map(n, kf, 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) { }

  § 22.5.4.1 867
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()
operator=(const 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>);
operator=(initializer_list<value_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);

pair<iterator, bool> try_emplace(const key_type& k, Args&& ... args);
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);

// 22.5.4.1 868

§ 22.5.4.1

868
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;
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 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);
4 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

```cpp
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));

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));

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 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);
template<class... Args>
iterator try_emplace(const_iterator hint, 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);
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 `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);
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 `unordered_map` from `std::move(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 `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.

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:
    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;

    // 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(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) { }

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&&) noexcept(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;

// 22.5.5.3, modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
template<class P> iterator insert(P&& obj);

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_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);

// 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;
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;
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>>,
    typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = 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>>, typename see below::size_type,
    Allocator)
-> unordered_multimap<Key, T, hash<Key>, equal_to<Key>, Allocator>;

// swap
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)));
}

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);

Effects: Equivalent to: return emplace(std::forward<P>(obj));

template<class P>
iterator insert(const_iterator hint, P&& obj);

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;
        }
    }
    return original_size - c.size();

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) { }
    }
}
```

§ 22.5.6.1
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) { }

template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const hasher& hf,
  const allocator_type& a)
  : unordered_set(f, l, n, hf, key_equal(), a) { }
unordered_set(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_set(il, 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();
unordered_set& operator=(const unordered_set&);
unordered_set& operator=(unordered_set&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_move_assignable_v<Hash> &&
    is_nothrow_move_assignable_v<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);
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& 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);

// 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;
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;
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_set<InputIterator, InputIterator, typename
see below::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>;

§ 22.5.6.1
template<class InputIterator, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_set<iter-value-type<InputIterator>,
    hash<iter-value-type<InputIterator>>,
    equal_to<iter-value-type<InputIterator>>,
    Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type,
    Hash, Allocator)
-> unordered_set<iter-value-type<InputIterator>, Hash,
    equal_to<iter-value-type<InputIterator>>,
    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>;

template<class T, class Hash, class Allocator>
unordered_set(initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> unordered_set<T, Hash, equal_to<T>, Allocator>;

// swap

//

4 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 [unord.set.cnstr]

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());

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());

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

```cpp
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);
```

1 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();
```

22.5.7 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.

```cpp
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
            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 allocator_type& a)  
  : unordered_multiset(n, hasher(), key_equal(), a) { }
unordered_multiset(size_type n, const hasher& hf, const allocator_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(const allocator_type& a)  
  : unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);  
unordered_multiset& operator=(unordered_multiset&&)  
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&  
    is_nothrow_move_assignable_v<Hash> &&  
    is_nothrow_move_assignable_v<Pred>);  
unordered_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;
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> 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(algorithm_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);
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);

// 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;
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 cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;
© ISO/IEC N4861

// 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 T, class Hash, class Allocator>
unordered_multiset(initializer_list<T>, typename see below::size_type, Hash, 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

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. \texttt{max_load_factor()} returns 1.0.

\textit{Complexity:} Constant.

\begin{verbatim}
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());
\end{verbatim}

\textit{Effects:} Constructs an empty \texttt{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. \texttt{max_load_factor()} returns 1.0.

\textit{Complexity:} Average case linear, worst case quadratic.

22.5.7.3 Erasure

\begin{verbatim}
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);
\end{verbatim}

\textit{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;
  }
}
return original_size - c.size();
\end{verbatim}

22.6 Container adaptors

22.6.1 In general

The headers \texttt{<queue>} and \texttt{<stack>} define the container adaptors \texttt{queue}, \texttt{priority_queue}, and \texttt{stack}.

The container adaptors each take a \texttt{Container} template parameter, and each constructor takes a \texttt{Container} reference argument. This container is copied into the \texttt{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 \(\text{Container}::\text{value_type}\).

For container adaptors, no \texttt{swap} function throws an exception unless that exception is thrown by the swap of the adaptor’s \texttt{Container} or \texttt{Compare} object (if any).

A deduction guide for a container adaptor shall not participate in overload resolution if any of the following are true:

1. \texttt{InputIterator} template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
2. \texttt{Compare} template parameter and a type that qualifies as an allocator is deduced for that parameter.
3. \texttt{Container} template parameter and a type that qualifies as an allocator is deduced for that parameter.
4. \texttt{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 17.10.1

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, class Container>
    compare_three_way_result_t<Container>
    operator<=>(const queue<T, Container>& x, const queue<T, Container>& y);

    template<class T, class Container = vector<T>,
             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>;
}
```

### 22.6.3 Header <stack> synopsis

```cpp
#include <compare> // see 17.11.1
#include <initializer_list> // see 17.10.1

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> //§ 22.6.3 887
    void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y)));

    template<class T, class Container, class Alloc>
    struct uses_allocator<stack<T, Container>, Alloc>;
}
```
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)));
template<class T, class Container, class Alloc>
struct uses_allocator<stack<T, Container>, Alloc>;

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()) {})
            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_type&&, const Alloc&);
            template<class Alloc> queue(const queue&, const Alloc&);

            template<[[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); }
            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<

    template<class Container, class Allocator>
    queue(Container, Allocator) -> queue<

    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 participate 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.

template<class T, class Container>
bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
6
Returns: x.c >= y.c.
template<class T, three_way_comparable Container>
    operator<=>(const queue<T, Container>& x, const queue<T, Container>& y);

    Returns: x.c <=> y.c.

22.6.4.5 Specialized algorithms

    Returns: x.c <=> y.c.

22.6.5 Class template priority_queue

22.6.5.1 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(const Compare&, 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(); }
            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>;

// no equality is provided

template<class T, class Container, class Compare, class Alloc>
struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
    : uses_allocator<Container, Alloc>::type { };

22.6.5.2 Constructors

priority_queue(const Compare& x, const 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);

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

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.

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, 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);

1    Effects: As if by:
          c.push_back(x);
          push_heap(c.begin(), c.end(), comp);

    void push(value_type&& x);

2    Effects: As if by:
          c.push_back(std::move(x));
          push_heap(c.begin(), c.end(), comp);

template<class... Args> void emplace(Args&&... args);

3    Effects: As if by:
          c.emplace_back(std::forward<Args>(args)...);
          push_heap(c.begin(), c.end(), comp);

    void pop();

4    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)));

1    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

Any sequence container supporting operations back(), push_back() and pop_back() can be used to instantiate stack. In particular, vector (22.3.11), list (22.3.10) and deque (22.3.8) can be used.

22.6.6.1 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;

    ...

§ 22.6.6.1
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& cont, const Alloc&);
template<class Alloc> stack(Container& cont, const Alloc&);
template<class Alloc> stack(const Container& cont, const Alloc& cont, const Alloc&);
template<class Alloc> stack(Container& cont, const Alloc& cont, 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 { };

22.6.6.2 Constructors [stack.cons]
explicit stack(const Container& cont);
Effects: Initializes c with cont.

explicit stack(Container& cont);
Effects: Initializes c with std::move(cont).

22.6.6.3 Constructors with allocators [stack.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 stack(const Alloc& a);
Effects: Initializes c with a.

template<class Alloc> stack(const container_type& cont, const Alloc& a);
Effects: Initializes c with cont as the first argument and a as the second argument.
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.

22.6.6.4 Operators

- 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, class Container>
  
  compare_three_way_result_t<Container> operator<=>(const stack<T, Container>& x, const stack<T, Container>& y);

  Returns: x.c <=> y.c.

22.6.6.5 Specialized algorithms

- 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

22.7.1 General

The header <span> defines the view span.

22.7.2 Header <span> synopsis

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;

§ 22.7.2 894
// 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

A span is a view over a contiguous sequence of objects, the storage of which is owned by some other object.

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<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(array<T, N>& arr) noexcept;
      template<class T, size_t N>
        constexpr span(array<T, N>& arr) noexcept;
      template<class R>
        constexpr explicit(see below) span(const span &r);
      constexpr span(const span& other) noexcept = default;
      ~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;

      // 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;
  }
}
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>>;}

template<class T, size_t N>
span(array<T, N>&) -> span<const T, N>;

template<class R>
span(R&&) -> span<remove_reference_t<ranges::range_reference_t<R>>>;

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: 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.

5 Effects: Initializes data_ with to_address(first) and size_ with count.

6 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: 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: 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: 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: 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: When and when ranges::data(r) and ranges::size(r) throw.

constexpr span(const span& other) noexcept = default;

Postconditions: other.size() == size() && other.data() == data().

template<class OtherElementType, size_t OtherExtent>
constexpr explicit(see below) span(const span<OtherElementType, OtherExtent>& s) noexcept;

Constraints:
— extent == dynamic_extent || OtherExtent == dynamic_extent || extent == OtherExtent is true, and

— is_convertible_v<OtherElementType(*[]), element_type(*[])> is true. [Note: The intent is to allow only qualification conversions of the OtherElementType to element_type. — end note]

20 Preconditions: If extent is not equal to dynamic_extent, then s.size() is equal to extent.

Effects: Constructs a span that is a view over the range [s.data(), s.data() + s.size()).

Postconditions: size() == s.size() && data() == s.data().

Remarks: The expression inside explicit is equivalent to:

extent != dynamic_extent && OtherExtent == dynamic_extent

constexpr span& operator=(const span& other) noexcept = default;

22 Postconditions: size() == other.size() && data() == other.data().

22.7.3.3 Deduction guides

template<class It, class EndOrSize>
span(It, EndOrSize) -> span<remove_reference_t<iter_reference_t<It>>;  

Constraints: It satisfies contiguous_iterator.

template<class R>
span(R&&) -> span<remove_reference_t<ranges::range_reference_t<R>>;  

Constraints: R satisfies ranges::contiguous_range.

22.7.3.4 Subviews

template<size_t Count> constexpr span<element_type, Count> first() const;

Mandates: Count <= Extent is true.

Preconditions: Count <= size() is true.

Effects: Equivalent to: return R{data(), Count}; where R is the return type.

template<size_t Count> constexpr span<element_type, Count> last() const;

Mandates: Count <= Extent is true.

Preconditions: Count <= size() is true.

Effects: Equivalent to: return R{data() + (size() - Count), Count}; where R is the return type.

template<size_t Offset, size_t Count = dynamic_extent>
constexpr span<element_type, see below> subspan() const;

Mandates: Offset <= Extent && (Count == dynamic_extent || Count <= Extent - Offset) is true.

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)

constexpr span<element_type, dynamic_extent> first(size_type count) const;

Preconditions: count <= size() is true.
Effects: Equivalent to: \(\text{return \{data(), count\};}\)

constexpr span<element_type, dynamic_extent> last(size_type count) const;

Preconditions: count \(\leq\) size() is true.

Effects: Equivalent to: \(\text{return \{data() + (size() - count), count\};}\)

constexpr span<element_type, dynamic_extent> subspan(
  size_type offset, size_type count = dynamic_extent) const;

Preconditions:
  offset \(\leq\) size() \&\& (count \(\leq\) dynamic_extent \| count \(\leq\) size() - offset)
  is true.

Effects: Equivalent to:
  \(\text{return \{data() + offset, count == dynamic_extent ? size() - offset : count\};}\)

22.7.3.5 Observers [span.obs]

constexpr size_type size() const noexcept;

Effects: Equivalent to: \(\text{return size_;}\)

constexpr size_type size_bytes() const noexcept;

Effects: Equivalent to: \(\text{return size() \ast sizeof(element_type);}\)

[[nodiscard]] constexpr bool empty() const noexcept;

Effects: Equivalent to: \(\text{return size() == 0;}\)

22.7.3.6 Element access [span.elem]

constexpr reference operator[](size_type idx) const;

Preconditions: idx \(<\) size() is true.

Effects: Equivalent to: \(\text{return *\{data() + idx\};}\)

constexpr reference front() const;

Preconditions: empty() is false.

Effects: Equivalent to: \(\text{return *data();}\)

constexpr reference back() const;

Preconditions: empty() is false.

Effects: Equivalent to: \(\text{return *(data() + (size() - 1));}\)

constexpr pointer data() const noexcept;

Effects: Equivalent to: \(\text{return data_;}\)

22.7.3.7 Iterator support [span.iterators]

using iterator = implementation-defined;

The type models contiguous_iterator (23.3.4.14), meets the Cpp17RandomAccessIterator requirements (23.3.5.6), and meets the requirements for constexpr iterators (23.3.1). All requirements on container iterators (22.2) apply to span::iterator as well.

constexpr iterator begin() const noexcept;

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;

Returns: An iterator which is the past-the-end value.
constexpr reverse_iterator rbegin() const noexcept;
 Effects: Equivalent to: return reverse_iterator(end());

constexpr reverse_iterator rend() const noexcept;
 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;
 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;
 Constraints: is_const_v<ElementType> is false.
 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 82: Iterators library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.3</td>
<td>&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

```cpp
#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 = requires { typename with_reference<T> };  // exposition only
  template<class T> concept dereferenceable = 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
    using iter_reference_t = decltype(*declval<T&>());

  namespace ranges {
    // 23.3.3, customization points
    inline namespace unspecified {
      // 23.3.3.1, ranges::iter_move
      inline constexpr unspecified iter_move = unspecified;
    }
  }
```

§ 23.2 901
// 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
// 23.4.1, 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 { }; 

// 23.4.2, 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);

// 23.4.3, range iterator operations
namespace ranges {
// 23.4.3.1, 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);

// 23.4.3.2, 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);

// 23.4.3.3, 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);
// 23.4.3.4, ranges::prev
template<bidirectional_iterator I>
constexpr I prev(I x);
template<bidirectional_iterator I>
constexpr I prev(I x, iter_difference_t<I> n);
template<bidirectional_iterator I>
constexpr I prev(I x, iter_difference_t<I> n, I bound);
}

// 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, 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+(typename reverse_iterator<Iterator>::difference_type n, const reverse_iterator<Iterator>& x);

template<class Iterator>
constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

template<class Iterator1, class Iterator2>
requires (!sized_sentinel_for<Iterator1, Iterator2>)
inline constexpr bool disable_sized_sentinel_for<reverse_iterator<Iterator1>, reverse_iterator<Iterator2>> = true;

// 23.5.2, insert iterators

template<class Container> class back_insert_iterator;

template<class Container>
constexpr back_insert_iterator<Container> back_inserter(Container& x);
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);

// 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;

// 23.5.5, default sentinels
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 sentinels
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 = char, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
const istream_iterator<T,charT,traits,Distance>& y);

template<class T, class charT = char, class traits = char_traits<charT>>
class ostream_iterator;
template<class charT, class traits = char_traits<charT>>
class istreambuf_iterator;
template<class charT, class traits = char_traits<charT>>
bool operator==(const istreambuf_iterator<charT,traits>& a,
const istreambuf_iterator<charT,traits>& b);

template<class T, class charT = char, class traits = char_traits<charT>>
class ostreambuf_iterator;

// 23.7, 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());
template<class C>
constexpr auto end(C& c) -> decltype(c.end());
template<class C>
constexpr auto end(const C& c) -> decltype(c.end());
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;
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 rend(C& c) -> decltype(c.rend());
template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);

§ 23.2 907
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

- input_iterator (23.3.4.9),
- output_iterator (23.3.4.10),
- forward_iterator (23.3.4.11),
- bidirectional_iterator (23.3.4.12),
- random_access_iterator (23.3.4.13), and
- 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: 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. [Example: After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a singular value of a pointer. —end example] 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. [Note: 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.

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 \(i\), \([i, s)\) denotes 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.

Iterators are called constexpr iterators if all operations provided to meet iterator category requirements
are constexpr functions. [Note: For example, the types “pointer to int” and reverse_iterator<int*> are
constexpr iterators. — end note]

23.3.2 Associated types  [iterator.assoc.types]
23.3.2.1 Incrementable traits  [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

iter_difference_t<\(WI\)>

be defined as the incrementable type’s difference type.

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; }

228) The sentinel denoting the end of a range may have the same type as the iterator denoting the beginning of the range, or a
different type.

229) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been
invalidated is undefined.
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

\[
\begin{align*}
\text{(2.1)} & \quad \text{\textcolor{blue}{incrementable\_traits}\(<R_I>\)::\text{difference\_type}} \quad \text{if iterator\_traits}\(<R_I>\) \quad \text{names a specialization generated from the primary template, and} \\
\text{(2.2)} & \quad \text{\textcolor{red}{iterator\_traits}\(<R_I>\)::\text{difference\_type}} \quad \text{otherwise.}
\end{align*}
\]

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 \textcolor{blue}{indirectly\_readable} concept (23.3.4.2), the type

\[
\text{iter\_value\_t}\(<R>\)
\]

be defined as the indirectly readable type’s value type.

```cpp
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<\textcolor{blue}{I}>*
struct indirectly\_readable\_traits<\textcolor{blue}{I}> \{ \\
    using value\_type = remove\_cv\_t<remove\_extent\_t<\textcolor{blue}{I}>>;
};

template<class I>
struct indirectly\_readable\_traits<const I> \:
    indirectly\_readable\_traits<\textcolor{blue}{I}> \{ \};

template<class T>
requires \textcolor{red}{requires \{ typename T::value\_type; \}}
struct indirectly\_readable\_traits<T> \:
    cond\_value\_type<typename T::value\_type> \{ \};

template<class T>
requires \textcolor{red}{requires \{ typename T::element\_type; \}}
struct indirectly\_readable\_traits<T> \:
    cond\_value\_type<typename T::element\_type> \{ \};
```
2 Let $R_I$ be `remove_cvref_t<`I`>`. The type `iter_value_t<`I`>` denotes

(2.1) — `indirectly_readable_traits<`R_I`>::value_type` if `iterator_traits<`R_I`>` names a specialization generated from the primary template, and

(2.2) — `iterator_traits<`R_I`>::value_type` otherwise.

3 Class template `indirectly_readable_traits` may be specialized on program-defined types.

4 [Note: Some legacy output iterators define a nested type named `value_type` that is an alias for `void`. These types are not `indirectly_readable` and have no associated value types. — end note]

5 [Note: Smart pointers like `shared_ptr<int>` are `indirectly_readable` and have an associated value type, but a smart pointer like `shared_ptr<void>` is not `indirectly_readable` and has no associated value type. — end note]

23.3.2.3 Iterator traits

1 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

`iterator_traits<`I`>::::iterator_category`

be defined as the iterator's iterator category. In addition, the types

`iterator_traits<`I`>::::pointer`

`iterator_traits<`I`>::::reference`

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 `decltype(a.operator->())` and `decltype(*a)`, respectively. The type `iterator_traits<`I`>::::pointer` shall be `void` for an iterator of class type `I` that does not support `operator->`. Additionally, in the case of an output iterator, the types

`iterator_traits<`I`>::::value_type`

`iterator_traits<`I`>::::difference_type`

`iterator_traits<`I`>::::reference`

may be defined as `void`.

2 The definitions in this subclause make use of the following exposition-only concepts:

```cpp
template<class T> using iter_value_t = see below;
```

Let $R_I$ be `remove_cvref_t<`I`>`. The type `iter_value_t<`I`>` denotes

(2.1) — `indirectly_readable_traits<`R_I`>::value_type` if `iterator_traits<`R_I`>` names a specialization generated from the primary template, and

(2.2) — `iterator_traits<`R_I`>::value_type` otherwise.

3 Class template `indirectly_readable_traits` may be specialized on program-defined types.

4 [Note: Some legacy output iterators define a nested type named `value_type` that is an alias for `void`. These types are not `indirectly_readable` and have no associated value types. — end note]

5 [Note: Smart pointers like `shared_ptr<int>` are `indirectly_readable` and have an associated value type, but a smart pointer like `shared_ptr<void>` is not `indirectly_readable` and has no associated value type. — end note]

23.3.2.3 Iterator traits

1 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

`iterator_traits<`I`>::::iterator_category`

be defined as the iterator's iterator category. In addition, the types

`iterator_traits<`I`>::::pointer`

`iterator_traits<`I`>::::reference`

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 `decltype(a.operator->())` and `decltype(*a)`, respectively. The type `iterator_traits<`I`>::::pointer` shall be `void` for an iterator of class type `I` that does not support `operator->`. Additionally, in the case of an output iterator, the types

`iterator_traits<`I`>::::value_type`

`iterator_traits<`I`>::::difference_type`

`iterator_traits<`I`>::::reference`

may be defined as `void`.

The definitions in this subclause make use of the following exposition-only concepts:

```cpp
template<class I>
concept cpp17-iterator =
    copyable<I> && requires(I i) {
    { *i } -> can-reference;
    { ++i } -> same_as<I&>;
    { *i++ } -> can-reference;
    };
```

```cpp
template<class I>
concept cpp17-input-iterator =
    cpp17-iterator<I> && equality_comparable<I> && requires(I i) {
    typename incrementable_traits<I>::::difference_type;
    typename indirectly_readable_traits<I>::::value_type;
    typename common_reference_t<iter_reference_t<I>&&,
        typename indirectly_readable_traits<I>::::value_type&>;
    typename common_reference_t<decltype(*i++)&&,
        typename indirectly_readable_traits<I>::::value_type&>;
    requires signed_integral<typename incrementable_traits<I>::::difference_type>;
    };
```

```cpp
template<class I>
concept cpp17-forward-iterator =
    cpp17-input-iterator<I> && constructible_from<I> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    same_as<remove_cvref_t<iter_reference_t<I>>>,
    typename indirectly_readable_traits<I>::::value_type &&
    requires(I i) {
```
The members of a specialization `iterator_traits<I>` generated from the `iterator_traits` primary template are computed as follows:

(3.1) — If `I` has valid (13.10.2) 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;
```

If the `qualified-id` `I::pointer` is valid and denotes a type, then `iterator_traits<I>::pointer` names that type, otherwise, it names `void`.

(3.2) — Otherwise, if `I` satisfies the exposition-only concept `cpp17-input-iterator`, `iterator_traits<I>` has the following publicly accessible members:

```cpp
using iterator_category = see below;
using value_type = typename indirectly_readable_traits<I>::value_type;
using difference_type = typename incrementable_traits<I>::difference_type;
using pointer = see below;
using reference = see below;
```

(3.2.1) — If the `qualified-id` `I::pointer` is valid and denotes a type, `pointer` names that type. Otherwise, `pointer` names `void`.

(3.2.2) — If the `qualified-id` `I::reference` is valid and denotes a type, `reference` names that type. Otherwise, `reference` names `iter_reference_t<I>`.

(3.2.3) — If the `qualified-id` `I::iterator_category` is valid and denotes a type, `iterator_category` names that type. Otherwise, `iterator_category` names:

- random_access_iterator_tag if `I` satisfies `cpp17-random-access-iterator`, or otherwise
- bidirectional_iterator_tag if `I` satisfies `cpp17-bidirectional-iterator`, or otherwise
- forward_iterator_tag if `I` satisfies `cpp17-forward-iterator`, or otherwise
- input_iterator_tag.

(3.3) — Otherwise, if `I` satisfies the exposition-only concept `cpp17-iterator`, then `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;
using pointer = void;
using reference = void;

If the qualified-id incrementable_traits<
I>::difference_type is valid and denotes a type, then
difference_type names that type; otherwise, it names void.

(3,4) — Otherwise, iterator_traits<
I> 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

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: To implement a generic reverse function, a C++ program can do the following:

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 points [iterator.cust]

23.3.3.1 ranges::iter_move [iterator.cust.move]

1 The name ranges::iter_move denotes a customization point object (16.4.2.2.6). 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

    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: 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.

§ 23.3.3.1
23.3.3.2 ranges::iter_swap

The name ranges::iter_swap denotes a customization point object (16.4.2.2.6) that exchanges the values (18.4.9) denoted by its arguments.

Let iter-exchange-move be the exposition-only function:

```cpp
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)));
```

Effects: Equivalent to:

```cpp
iter_value_t<X> old_value(iter_move(x));
*x = iter_move(y);
return old_value;
```

The expression ranges::iter_swap(E1, E2) for subexpressions E1 and E2 is expression-equivalent to:

1. (void)iter_swap(E1, E2), if either E1 or E2 has class or enumeration type and iter_swap(E1, E2) is a well-formed expression with overload resolution performed in a context that includes the declaration

```cpp
template<class I1, class I2>
void iter_swap(I1, I2) = delete;
```

and does not include a declaration of 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.

2. Otherwise, if the types of E1 and E2 each model indirectly_readable, and if the reference types of E1 and E2 model swappable_with (18.4.9), then ranges::swap(*E1, *E2).

3. Otherwise, if the types T1 and T2 of E1 and E2 model indirectly_movable_storable<T1, T2> and indirectly_movable_storable<T2, T1>, then (void)(*E1 = iter-exchange-move(E2, E1)), except that E1 is evaluated only once.

4. Otherwise, ranges::iter_swap(E1, E2) is ill-formed. [Note: This case can result in substitution failure when 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 ITER_TRAITS(I) denote the type I if iterator_traits<I> names a specialization generated from the primary template. Otherwise, ITER_TRAITS(I) denotes iterator_traits<I>.

1. If the qualified-id ITER_TRAITS(I)::iterator_concept is valid and names a type, then ITER_CONCEPT(I) denotes that type.

2. Otherwise, if the qualified-id ITER_TRAITS(I)::iterator_category is valid and names a type, then ITER_CONCEPT(I) denotes that type.

3. Otherwise, if iterator_traits<I> names a specialization generated from the primary template, then ITER_CONCEPT(I) denotes random_access_iterator_tag.

4. Otherwise, ITER_CONCEPT(I) does not denote a type.

[Note: ITER_TRAITS enables independent syntactic determination of an iterator’s category and concept. —end note]
bool operator!=(I) const;
};

iterator_traits<I>::iterator_category denotes input_iterator_tag, and ITER_CONCEPT(I) denotes random_access_iterator_tag. —end example]

23.3.4.2 Concept indirectly_readable [iterator.concept.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: 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 [iterator.concept.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<Out>(t); // not required to be equality-preserving
        const_cast<const iter_reference_t<Out>&&>(*o) =
            std::forward<Out>(t); // not required to be equality-preserving
        const_cast<const iter_reference_t<Out>&&>(*std::forward<Out>(o)) =
            std::forward<Out>(t); // not required to be equality-preserving
    };
```

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.

After evaluating any above assignment expression, o is not required to be dereferenceable.

If E is an xvalue (7.2.1), the resulting state of the object it denotes is valid but unspecified (16.5.5.16).

[Note: 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]

(2.1) [Note: 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 [iterator.concept.winc]

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.
A type \( I \) is an \textit{integer-class type} if it is in a set of implementation-defined class types that behave as integer types do, as defined in below.

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 \textit{signed-integer-class type}; otherwise, it is an \textit{unsigned-integer-class type}.

For every integer-class type \( I \), let \( B(I) \) be a hypothetical extended integer type of the same signedness with the smallest width (6.8.1) capable of representing the same range of values. The width of \( I \) is equal to the width of \( B(I) \).

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.

Expression 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.

An expression \( E \) of integer-class type \( I \) is contextually convertible to \texttt{bool} as if by \texttt{bool(E != I(0))}.

All integer-class types model \textit{regular} (18.6) and \textit{totally_ordered} (18.5.4).

A value-initialized object of integer-class type has value 0.

For every (possibly cv-qualified) integer-class type \( I \), \texttt{numeric_limits<}I\texttt{>::is_specialized} is specialized such that:

\begin{enumerate}
\item \texttt{numeric_limits<}I\texttt{>::is_signed} is true if and only if \( I \) is a signed-integer-class type,
\item \texttt{numeric_limits<}I\texttt{>::is_integer} is true,
\item \texttt{numeric_limits<}I\texttt{>::is_exact} is true,
\item \texttt{numeric_limits<}I\texttt{>::digits} is equal to the width of the integer-class type,
\item \texttt{numeric_limits<}I\texttt{>::digits10} is equal to \texttt{static_cast<int>(digits * log10(2))}, and
\item \texttt{numeric_limits<}I\texttt{>::min()} and \texttt{numeric_limits<}I\texttt{>::max()} return the lowest and highest representable values of \( I \), respectively, and \texttt{numeric_limits<}I\texttt{>::lowest()} returns \texttt{numeric_limits<}I\texttt{>::min()}.
\end{enumerate}
A type \( I \) is integer-like if it models \( \text{integral}<I> \) or if it is an integer-class type. A type \( I \) is signed-integer-like if it models \( \text{signed_integral}<I> \) or if it is a signed-integer-class type. A type \( I \) is unsigned-integer-like if it models \( \text{unsigned_integral}<I> \) or if it is an unsigned-integer-class type.

\( \text{is-integer-like}<I> \) is true if and only if \( I \) is an integer-like type. \( \text{is-signed-integer-like}<I> \) is true if and only if \( I \) is a signed-integer-like type.

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 \( \text{weakly_incrementable}<I> \) only if

\begin{enumerate}
\item The expressions \( ++i \) and \( i++ \) have the same domain.
\item If \( i \) is incrementable, then both \( ++i \) and \( i++ \) advance \( i \) to the next element.
\item If \( i \) is incrementable, then \( \text{addressof}(++i) \) is equal to \( \text{addressof}(i) \).
\end{enumerate}

[Note: For \text{weakly_incrementable} types, \( a \) equals \( b \) does not imply that \( ++a \) equals \( ++b \). (Equality does not guarantee the substitution property or referential transparency.) Algorithms on weakly incrementable types should not guarantee to pass through the same incrementable value twice. They should be single-pass algorithms. These algorithms can be used with istreams as the source of the input data through the \text{istream_iterator} class template. — end note]

23.3.4.5 Concept \text{incrementable} \hspace{1cm} \text{[iterator.concept-inc]}

The \text{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 \text{equality_comparable}. [Note: This supersedes the annotations on the increment expressions in the definition of \text{weakly_incrementable}. — end note]

\text{template<class I>}
\text{concept incrementable =}
\text{regular<I> \&\&}
\text{weakly_incrementable<I> \&\&}
\text{requires(I i) \{}
\text{ \{ i++ \} \rightarrow same_as<I>;}
\text{\};}

2 Let \( a \) and \( b \) be incrementable objects of type \( I \). I models \text{incrementable} only if

\begin{enumerate}
\item If \( \text{bool}(a == b) \) then \( \text{bool}(a++ == b) \).
\item If \( \text{bool}(a == b) \) then \( \text{bool}(((\text{void})a++, a) == ++b) \).
\end{enumerate}

[Note: 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 \text{incrementable}. — end note]

23.3.4.6 Concept \text{input_or_output_iterator} \hspace{1cm} \text{[iterator.concept.iterator]}

The \text{input_or_output_iterator} concept forms the basis of the iterator concept taxonomy; every iterator models \text{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) values, or to provide a richer set of iterator movements (23.3.4.11, 23.3.4.12, 23.3.4.13).

\text{template<class I>}
\text{concept input_or_output_iterator =}
\text{requires(I i) \{}
\text{ \{ *i \} \rightarrow can-reference;}
\text{\} \&\&}
\text{weakly_incrementable<I>;}

2 [Note: Unlike the Cpp17Iterator requirements, the \text{input_or_output_iterator} concept does not require copyability. — end note]

23.3.4.7 Concept \text{sentinel_for} \hspace{1cm} \text{[iterator.concept.sentinel]}

The \text{sentinel_for} concept specifies the relationship between an \text{input_or_output_iterator} type and a \text{semiregular} type whose values denote a range.
Let $s$ and $i$ be values of type $S$ and $I$ such that $[i, s)$ denotes a range. Types $S$ and $I$ model $\text{sentinel\_for}<S, I>$ only if

(2.1) $i == s$ is well-defined.

(2.2) If $\text{bool}(i != s)$ 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

The $\text{sized\_sentinel\_for}$ concept specifies requirements on an $\text{input\_or\_output\_iterator}$ type $I$ and a corresponding $\text{sentinel\_for}\langle I\rangle$ that allow the use of the $-$ operator to compute the distance between them in constant time.

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 $\text{bool}(i == s)$ be true. $S$ and $I$ model $\text{sized\_sentinel\_for}<S, I>$ only if

(2.1) If $N$ is representable by $\text{iter\_difference\_t}\langle I\rangle$, then $s - i$ is well-defined and equals $N$.

(2.2) If $-N$ is representable by $\text{iter\_difference\_t}\langle I\rangle$, then $i - s$ is well-defined and equals $-N$.

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.
23.3.4.10 Concept output_iterator  

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.  

[Note: Output iterators are not required to model **equality_comparable**. — end note]

```cpp
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
  }
```

2 Let E be an expression such that `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:

- `*i = E;
- ++i;

[Note: Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be single-pass algorithms. — end note]

23.3.4.11 Concept forward_iterator  

The **forward_iterator** concept adds copyability, equality comparison, and the multi-pass guarantee, specified below.

```cpp
template<class I>
concept forward_iterator =
  input_iterator<I> &&
  derived_from<ITER_CONCEPT(I), forward_iterator_tag> &&
  incrementable<I> &&
  sentinel_for<I, I>;
```

2 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: Value-initialized iterators behave as if they refer past the end of the same empty sequence. — end note]

3 Pointers and references obtained from a forward iterator into a range `[i, s)` shall remain valid while `[i, s)` continues to denote a range.

4 Two dereferenceable iterators a and b of type X offer the **multi-pass guarantee** if:

- (4.1) `a == b` implies `++a == ++b` and
- (4.2) The expression `((void)[](X x){++x;}(a), *a)` is equivalent to the expression `*a`.

[Note: The requirement that `a == b` implies `++a == ++b` and 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.

```cpp
template<class I>
concept bidirectional_iterator =
  forward_iterator<I> &&
  derived_from<ITER_CONCEPT(I), bidirectional_iterator_tag> &&
  requires(I i) {
    { --i } -> same_as<I&>;
    { i-- } -> same_as<I>;
  }
```

2 A bidirectional iterator r is decrementable if and only if there exists some q such that `++q == r`. Decrementable iterators r shall be in the domain of the expressions `--r` and `r--`.

3 Let a and b be equal objects of type I. I models **bidirectional_iterator** only if:

- (3.1) If a and b are decrementable, then all of the following are `true`:
  ```cpp
  addressof(--a) == addressof(a)
  ```

§ 23.3.4.12
(3.1.2)  — bool(a-- == b)
(3.1.3)  — after evaluating both a-- and --b, bool(a == b) is still true
(3.1.4)  — bool(++(--a) == b)
(3.2)  — If a and b are incrementable, then bool(--(++a) == b).

23.3.4.13 Concept random_access_iterator

The random_access_iterator concept adds support for constant-time advancement with ++$+, --$, and $-$, as well as the computation of distance in constant time with $\cdot$. Random access iterators also support array notation via subscripting.

```cpp
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>>;
    };
```

2 Let a and b be valid iterators of type I such that b is reachable from a after $n$ applications of ++$+, 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

The contiguous_iterator concept provides a guarantee that the denoted elements are stored contiguously in memory.

```cpp
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>>>;
    };
```

2 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
23.3.5 C++17 iterator requirements

1 In the following sections, \( a \) and \( b \) denote values of type \( X \) or const \( X \), difference_type and reference refer to the types \( \text{iterator_traits}<X>::\text{difference_type} \) and \( \text{iterator_traits}<X>::\text{reference} \), respectively; \( n \) denotes a value of difference_type, \( u \), tmp, and \( m \) denote identifiers, \( r \) denotes a value of \( X \), \( t \) denotes a value of some type that is writable to the output iterator. [Note: For an iterator type \( X \) there must be an instantiation of \( \text{iterator_traits}<X> \) (23.3.2.3). — end note]

23.3.5.1 Cpp17Iterator

1 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 (23.3.5.2) or write (23.3.5.3) values, or to provide a richer set of iterator movements (23.3.5.4, 23.3.5.5, 23.3.5.6).

2 A type \( X \) meets the Cpp17Iterator requirements if:

(2.1) \( X \) meets the Cpp17CopyConstructible, Cpp17CopyAssignable, and Cpp17Destructible requirements (16.5.3.1) and lvalues of type \( X \) are swappable (16.5.3.2), and

(2.2) iterator_traits\( <X>::\text{difference_type} \) is a signed integer type or void, and

(2.3) the expressions in Table 84 are valid and have the indicated semantics.

<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>unspecified</td>
<td>Preconditions: ( r ) is dereferenceable.</td>
<td></td>
</tr>
<tr>
<td>++( r )</td>
<td>( X )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23.3.5.2 Input iterators

1 A class or pointer type \( X \) meets the requirements of an input iterator for the value type \( T \) if \( X \) meets the Cpp17Iterator (23.3.5.1) and Cpp17EqualityComparable (Table 25) requirements and the expressions in Table 85 are valid and have the indicated semantics.

2 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: 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>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</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>Preconditions: ( a ) is dereferenceable.</td>
<td>The expression ( \text{void} )*( a ), ( *a ) is equivalent to ( \ast a ). If ( a == b ) and ( (a, b) ) is in the domain of ( == ) then ( *a ) is equivalent to ( \ast b ).</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><code>a-&gt;m</code></td>
<td><code>(*a).m</code></td>
<td></td>
<td>Preconditions: <code>a</code> is dereferenceable.</td>
</tr>
<tr>
<td><code>++r</code></td>
<td><code>X&amp;</code></td>
<td></td>
<td>Preconditions: <code>r</code> is dereferenceable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Postconditions: <code>r</code> is dereferenceable or <code>r</code> is past-the-end; any copies of the previous value of <code>r</code> are no longer required to be dereferenceable nor to be in the domain of <code>==</code>.</td>
</tr>
<tr>
<td><code>(void)r++</code></td>
<td></td>
<td></td>
<td>equivalent to <code>(void)++r</code></td>
</tr>
</tbody>
</table>

*Note: For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be single pass algorithms. Value type `T` is not required to be a `Cpp17CopyAssignable` type (Table 31). These algorithms can be used with streams as the source of the input data through the `istream_iterator` class template. — end note*

### 23.3.5.3 Output iterators

A class or pointer type `X` meets the requirements of an output iterator if `X` meets the `Cpp17Iterator` requirements (23.3.5.1) and the expressions in Table 86 are valid and have the indicated semantics.

### Table 86: `Cpp17OutputIterator` 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><code>*r = o</code></td>
<td>result is not used</td>
<td></td>
<td>Remarks: After this operation <code>r</code> is not required to be dereferenceable. Postconditions: <code>r</code> is incrementable.</td>
</tr>
<tr>
<td><code>++r</code></td>
<td><code>X&amp;</code></td>
<td></td>
<td><code>addressof(r) == addressof(++r)</code>. Remarks: After this operation <code>r</code> is not required to be dereferenceable. Postconditions: <code>r</code> is incrementable.</td>
</tr>
<tr>
<td><code>r++</code></td>
<td>convertible to <code>const X&amp;</code></td>
<td>`{ X tmp = r; ++r; return tmp; }</td>
<td>Remarks: After this operation <code>r</code> is not required to be dereferenceable. Postconditions: <code>r</code> is incrementable.</td>
</tr>
<tr>
<td><code>*r++ = o</code></td>
<td>result is not used</td>
<td></td>
<td>Remarks: After this operation <code>r</code> is not required to be dereferenceable. Postconditions: <code>r</code> is incrementable.</td>
</tr>
</tbody>
</table>
2. "Note: 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. Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be single-pass algorithms. Equality and inequality might not be defined. —end note"

23.3.5.4  Forward iterators

A class or pointer type X meets the requirements of a forward iterator if

1. X meets the Cpp17InputIterator requirements (23.3.5.2),
2. X meets the Cpp17DefaultConstructible requirements (16.5.3.1),
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,
4. the expressions in Table 87 are valid and have the indicated semantics, and
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: Value-initialized iterators behave as if they refer past the end of the same empty sequence. —end note]

2. 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 (void)++X(a), *a is equivalent to the expression *a.

[Note: 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)  [tab:forwarditerator]

<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>convertible to</td>
<td>X tmp = r; ++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.

23.3.5.5  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.

Table 88: Cpp17BidirectionalIterator requirements (in addition to Cpp17ForwardIterator)  [tab:bidirectionaliterator]

<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>Preconditions: there exists s such that r == ++s. Postconditions: r is dereferenceable. --(++r) == r. --r == --s implies r == s. addressof(r) == addressof(--r).</td>
<td></td>
</tr>
</tbody>
</table>

§ 23.3.5.5 923
### Table 88: Cpp17BidirectionalIterator requirements (in addition to Cpp17ForwardIterator) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>r--</td>
<td>convertible to</td>
<td>{ X tmp = r; --r; return tmp; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*r--</td>
<td>reference</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 [Note: Bidirectional iterators allow algorithms to move iterators backward as well as forward. —end note]

### 23.3.5.6 Random access iterators [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.

### Table 89: Cpp17RandomAccessIterator requirements (in addition to Cpp17BidirectionalIterator) [tab:randomaccessiterator]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>r += n</td>
<td>X&amp;</td>
<td>{ difference_type m = n; if (m &gt;= 0) while (m--) ++r; else while (m++) --r; return r; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a + n</td>
<td>X</td>
<td>{ X tmp = a; return tmp += n; }</td>
<td>a + n == n + a.</td>
<td></td>
</tr>
<tr>
<td>n + a</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r -= n</td>
<td>X&amp;</td>
<td>return r += -n;</td>
<td>Preconditions: the absolute value of n is in the range of representable values of difference_type.</td>
<td></td>
</tr>
<tr>
<td>a - n</td>
<td>X</td>
<td>{ X tmp = a; return tmp -= n; }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b - a</td>
<td>difference_type</td>
<td>return n</td>
<td>Preconditions: there exists a value n of type difference_type such that a + n == b. b == a + (b - a).</td>
<td></td>
</tr>
<tr>
<td>a[n]</td>
<td>convertible to reference</td>
<td>*(a + n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &lt; b</td>
<td>contextually convertible to bool</td>
<td>b - a &gt; 0</td>
<td>&lt; is a total ordering relation</td>
<td></td>
</tr>
<tr>
<td>a &gt; b</td>
<td>contextually convertible to bool</td>
<td>b &lt; a</td>
<td>&gt; is a total ordering relation opposite to &lt;.</td>
<td></td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>contextually convertible to bool</td>
<td>!(a &lt; b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>contextually convertible to bool</td>
<td>!(a &gt; b)</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 (20.14.2) as arguments.

23.3.6.2 Indirect callables

The indirect callable concepts are used to constrain those algorithms that accept callable objects (20.14.2) as arguments.

```cpp
namespace std {
    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 indirectly 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_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

1 Class template projected is used to constrain algorithms that accept callable objects and projections (16.3.22). 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
    };

    template<weakly_incrementable I, class Proj>
    struct incrementable_traits<projected<I, Proj>> {
        using difference_type = iter_difference_t<I>;
    };
}

23.3.7 Common algorithm requirements

23.3.7.1 General

1 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.

2 [Note: 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 (20.14.8). — end note]

23.3.7.2 Concept indirectly_movable

1 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>>;

2 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> &&
        indirectly_writable<Out, iter_value_t<In>> &&
        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 In. In and Out model indirectly_movable_storable<In, Out> only if after the initialization of the object obj in

    iter_value_t<In> obj(ranges::iter_move(i));
obj is equal to the value previously denoted by \(*i\). If \texttt{iter\_rvalue\_reference\_t<In>} is an rvalue reference type, the resulting state of the value denoted by \(*i\) is valid but unspecified (16.5.5.16).

23.3.7.3 Concept \texttt{indirectly\_copyable} [alg.req.ind.copy]

1 The \texttt{indirectly\_copyable} concept specifies the relationship between a \texttt{indirectly\_readable} type and a \texttt{indirectly\_writable} type between which values may be copied.

```cpp
template<class In, class Out>
concept indirectly_copyable =
    indirectly_readable<In> &&
    indirectly_writable<Out, iter_reference_t<In>>;
```

2 The \texttt{indirectly\_copyable\_storable} concept augments \texttt{indirectly\_copyable} with additional requirements enabling the transfer to be performed through an intermediate object of the \texttt{indirectly\_readable} type’s value type. It also requires the capability to make copies of values.

```cpp
template<class In, class Out>
concept indirectly_copyable_storable =
    indirectly_copyable<In, Out> &&
    indirectly_writable<Out, iter_value_t<In>&> &&
    indirectly_writable<Out, const iter_value_t<In>&> &&
    indirectly_writable<Out, iter_value_t<In>&&> &&
    indirectly_writable<Out, const iter_value_t<In>&&> &&
    copyable<iter_value_t<In>> &&
    constructible_from<iter_value_t<In>, iter_reference_t<In>> &&
    assignable_from<iter_value_t<In>&, iter_reference_t<In>>;
```

3 Let \(i\) be a dereferenceable value of type \(In\). \(In\) and \(Out\) model \texttt{indirectly\_copyable\_storable<In, Out>} only if after the initialization of the object \(\texttt{obj}\) in

```cpp
iter_value_t<In> obj(*i);
```

\(\texttt{obj}\) is equal to the value previously denoted by \(*i\). If \texttt{iter\_reference\_t<In>} is an rvalue reference type, the resulting state of the value denoted by \(*i\) is valid but unspecified (16.5.5.16).

23.3.7.4 Concept \texttt{indirectly\_swappable} [alg.req.ind.swap]

1 The \texttt{indirectly\_swappable} concept specifies a swappable relationship between the values referenced by two \texttt{indirectly\_readable} types.

```cpp
template<class I1, class I2 = I1>
concept indirectly_swappable =
    indirectly_readable<I1> && indirectly_readable<I2> &&
    requires(const I1 i1, const I2 i2) {
        ranges::iter_swap(i1, i1);
        ranges::iter_swap(i2, i2);
        ranges::iter_swap(i1, i2);
        ranges::iter_swap(i2, i1);
    };
```

23.3.7.5 Concept \texttt{indirectly\_comparable} [alg.req.ind.cmp]

1 The \texttt{indirectly\_comparable} concept specifies the common requirements of algorithms that compare values from two different sequences.

```cpp
template<class I1, class I2, class R, class P1 = identity,
    class P2 = identity>
concept indirectly_comparable =
    indirect_binary_predicate<R, projected<I1, P1>, projected<I2, P2>>;
```

23.3.7.6 Concept \texttt{permutable} [alg.req.permutable]

1 The \texttt{permutable} concept specifies the common requirements of algorithms that reorder elements in place by moving or swapping them.

```cpp
template<class I>
concept permutable =
    forward_iterator<I> &&
    indirectly_movable_storable<I, I> &&
    indirectly_swappable<I, I>;
```
23.3.7.7 Concept mergeable
1 The mergeable concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by copying elements.

```cpp
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
1 The sortable concept specifies the common requirements of algorithms that permute sequences into ordered sequences (e.g., sort).

```cpp
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
1 To simplify the use of iterators, the library provides several classes and functions.

23.4.1 Standard 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).

```cpp
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: For a program-defined iterator BinaryTreeIterator, it could be included into the bidirectional iterator category by specializing the iterator_traits template:

```cpp
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&;
};
```

—end example]

3 [Example: 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());
```
template<class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
   bidirectional_iterator_tag) {
    // more generic, but less efficient algorithm
}

template<class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
   random_access_iterator_tag) {
    // more efficient, but less generic algorithm
}

—end example

23.4.2 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.

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.

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.

template<class InputIterator>
constexpr InputIterator next(InputIterator x,
   typename iterator_traits<InputIterator>::difference_type n = 1);

Effects: Equivalent to: advance(x, n); return x;

template<class BidirectionalIterator>
constexpr BidirectionalIterator prev(BidirectionalIterator x,
   typename iterator_traits<BidirectionalIterator>::difference_type n = 1);

Effects: Equivalent to: advance(x, -n); return x;

23.4.3 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. [Example: 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 this subclause are not found by argument-dependent name lookup (6.5.2). When found by unqualified (6.5.1) name lookup for the postfix-expression in a function call (7.6.1.2), they inhibit argument-dependent name lookup.

[Example:

```cpp
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1,2,3};
```]
```cpp
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 (13.7.6.2) 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 this subclause is unspecified, except where explicitly stated otherwise.

### 23.4.3.1 ranges::advance

```cpp
template<input_or_output_iterator I>
constexpr void ranges::advance(I& i, iter_difference_t<I> n);
```

1 **Preconditions:** If I does not model `bidirectional_iterator`, n is not negative.
2 **Effects:**
   - If I models `random_access_iterator`, equivalent to `i += n`.
   - Otherwise, if n is non-negative, increments i by n.
   - Otherwise, decrements i by -n.

```cpp
template<input_or_output_iterator I, sentinel_for<I> S>
constexpr void ranges::advance(I& i, S bound);
```

3 **Preconditions:** [i, bound) denotes a range.
4 **Effects:**
   - If I and S model `assignable_from<I&, S>`, equivalent to `i = std::move(bound)`.
   - Otherwise, if S and I model `sized_sentinel_for<S, I>`, equivalent to `ranges::advance(i, bound - i)`.
   - Otherwise, while `bool(i != bound)` is true, increments i.

```cpp
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);
```

5 **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>`.
6 **Effects:**
   - If S and I model `sized_sentinel_for<S, I>`:
     1. If `|n| ≥ |bound - i|`, equivalent to `ranges::advance(i, bound)`.
     2. Otherwise, equivalent to `ranges::advance(i, n)`.
   - Otherwise,
     1. If n is non-negative, while `bool(i != bound)` is true, increments i but at most `n` times.
     2. Otherwise, while `bool(i != bound)` is true, decrements i but at most `-n` times.
7 **Returns:** `n - M`, where M is the difference between the ending and starting positions of i.

### 23.4.3.2 ranges::distance

```cpp
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>` 
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`.

§ 23.4.3.2 930
template<range R>
constexpr range_difference_t<R> ranges::distance(R&& r);

Effects: If R models sized_range, equivalent to:
        return static_cast<range_difference_t<R>>(ranges::size(r));  // 24.3.9

Otherwise, equivalent to:
        return ranges::distance(ranges::begin(r), ranges::end(r));  // 24.3

23.4.3.3 ranges::next

template<input_or_output_iterator I>
constexpr I ranges::next(I x);

Effects: Equivalent to: ++x; return x;

template<input_or_output_iterator I>
constexpr I ranges::next(I x, iter_difference_t<I> n);

Effects: Equivalent to: ranges::advance(x, n); return x;

template<input_or_output_iterator I, sentinel_for<I> S>
constexpr I ranges::next(I x, S bound);

Effects: Equivalent to: ranges::advance(x, bound); return x;

template<input_or_output_iterator I, sentinel_for<I> S>
constexpr I ranges::next(I x, iter_difference_t<I> n, S bound);

Effects: Equivalent to: ranges::advance(x, n, bound); return x;

23.5 Iterator adaptors

23.5.1 Reverse iterators

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.

23.5.1.1 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 derived_from<random_access_iterator_tag>, and
— iterator_traits<Iterator>::iterator_category otherwise.

23.5.1.2 Requirements [reverse.iter.requirements]

1 The template parameter Iterator shall either meet the requirements of a Cpp17BidirectionalIterator (23.3.5.5) or model bidirectional_iterator (23.3.4.12).

2 Additionally, Iterator shall either meet the requirements of a Cpp17RandomAccessIterator (23.3.5.6) or model random_access_iterator (23.3.4.13) if the definitions of any of the members
— operator+, operator-, operator+=, operator-= (23.5.1.6), or
— operator[] (23.5.1.5),

or the non-member operators (23.5.1.7)
— operator<, operator>, operator<=, operator>=, operator-, or operator+ (23.5.1.8)

are instantiated (13.9.1).

23.5.1.3 Construction and assignment [reverse.iter.cons]

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.

§ 23.5.1.3 932
constexpr explicit reverse_iterator(Iterator x);

Effects: Initializes current with x.

```cpp
template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);
```

Effects: Initializes current with u.current.

```cpp
template<class U>
constexpr reverse_iterator&
operator=(const reverse_iterator<U>& u);
```

Effects: Assigns u.base() to current.

Returns: *this.

23.5.1.4 Conversion

```cpp
constexpr Iterator base() const; // explicit
```

Returns: current.

23.5.1.5 Element access

```cpp
constexpr reference operator*() const;
```

Effects: As if by:

```cpp
Iterator tmp = current;
return *--tmp;
```

```cpp
constexpr pointer operator->() const
requires (is_pointer_v<Iterator> ||
requires (const Iterator i) { i.operator->(); });
```

Effects:

- (2.1) If Iterator is a pointer type, equivalent to: return prev(current);
- (2.2) Otherwise, equivalent to: return prev(current).operator->();

```cpp
constexpr unspecified operator[](difference_type n) const;
```

Returns: current[-n-1].

23.5.1.6 Navigation

```cpp
constexpr reverse_iterator operator+(difference_type n) const;
```

Returns: reverse_iterator(current-n).

```cpp
constexpr reverse_iterator operator-(difference_type n) const;
```

Returns: reverse_iterator(current+n).

```cpp
constexpr reverse_iterator& operator++();
```

Effects: As if by: --current;

Returns: *this.

```cpp
constexpr reverse_iterator operator++(int);
```

Effects: As if by:

```cpp
reverse_iterator tmp = *this;
--current;
return tmp;
```

```cpp
constexpr reverse_iterator& operator--();
```

Effects: As if by ++current.

Returns: *this.

§ 23.5.1.6
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.

### 23.5.1.7 Comparisons

[reverse.iter.cmp]

```cpp
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().

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, 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);

Returns: y.base() <=> x.base().
[Note: The argument order in the Returns: element is reversed because this is a reverse iterator. — end note]

23.5.1.8 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+(
    typename reverse_iterator<Iterator>::difference_type 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(see below);

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&>()));

friend constexpr void
iter_swap(const reverse_iterator& x,
          const reverse_iterator<Iterator2>& y) noexcept(see below);

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

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,

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

§ 23.5.2
meet the requirements of output iterators. \texttt{operator*} returns the insert iterator itself. The assignment \texttt{operator=(const T& x)} is defined on insert iterators to allow writing into them, it inserts \texttt{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. \texttt{back_insert_iterator} inserts elements at the end of a container, \texttt{front_insert_iterator} inserts elements at the beginning of a container, and \texttt{insert_iterator} inserts elements where the iterator points to in a container. \texttt{back_inserter}, \texttt{front_inserter}, and \texttt{inserter} are three functions making the insert iterators out of a container.

23.5.2.1 Class template \texttt{back_insert_iterator}  

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);
    };
}

23.5.2.1.1 Operations  

constexpr explicit back_insert_iterator(Container& x);
\begin{enumerate}
\item \texttt{Effects:} Initializes \texttt{container} with \texttt{addressof(x)}.
\item constexpr back_insert_iterator& operator=(const typename Container::value_type& value);
\item \texttt{Effects:} As if by: \texttt{container->push_back(value)};
\item \texttt{Returns:} \texttt{*this}.
\item constexpr back_insert_iterator& operator=(typename Container::value_type&& value);
\item \texttt{Effects:} As if by: \texttt{container->push_back(std::move(value))};
\item \texttt{Returns:} \texttt{*this}.
\item constexpr back_insert_iterator& operator*();
\item \texttt{Returns:} \texttt{*this}.
\item constexpr back_insert_iterator& operator++();
\item constexpr back_insert_iterator& operator++(int);
\item \texttt{Returns:} \texttt{*this}.
\end{enumerate}

23.5.2.1.2 \texttt{back_inserter}  

template<class Container>
    constexpr back_insert_iterator<Container> back_inserter(Container& x);
\begin{enumerate}
\item \texttt{Returns:} \texttt{back_insert_iterator<Container>(x)}.
\end{enumerate}
23.5.2.2 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);
    };
}

23.5.2.2.1 Operations

constexpr explicit front_insert_iterator(Container& x);
1 Effects: Initializes container with addressof(x).

constexpr front_insert_iterator& operator=(const typename Container::value_type& value);
2 Effects: As if by: container->push_front(value);
3 Returns: *this.

constexpr front_insert_iterator& operator=(typename Container::value_type&& value);
4 Effects: As if by: container->push_front(std::move(value));
5 Returns: *this.

constexpr front_insert_iterator& operator*();
6 Returns: *this.

constexpr front_insert_iterator& operator++();
7 Returns: *this.

23.5.2.2.2 front_inserter

template<class Container>
    constexpr front_insert_iterator<Container> front_inserter(Container& x);
1 Returns: front_insert_iterator<Container>(x).

23.5.2.3 Class template insert_iterator

namespace std {
    template<class Container>
    class insert_iterator {
    protected:
        Container* container = nullptr;
        ranges::iterator_t<Container> iter = ranges::iterator_t<Container>();
    };
}

§ 23.5.2.3 937
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);

};

23.5.2.3.1 Operations

constexpr insert_iterator(Container& x, ranges::iterator_t<Container> i);

1 Effects: Initializes container with addressof(x) and iter with i.

constexpr insert_iterator& operator=(const typename Container::value_type& value);

2 Effects: As if by:
   iter = container->insert(iter, value);
   ++iter;

3 Returns: *this.

constexpr insert_iterator& operator=(typename Container::value_type&& value);

4 Effects: As if by:
   iter = container->insert(iter, std::move(value));
   ++iter;

5 Returns: *this.

constexpr insert_iterator& operator*();

6 Returns: *this.

constexpr insert_iterator& operator++();
constexpr insert_iterator& operator++(int);

7 Returns: *this.

23.5.2.3.2 inserter

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

Class template move_iterator is an iterator adaptor with the same behavior as the underlying iterator except that its indirect 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:

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]

23.5.3.1 Class template move_iterator

namespace std {

    template<class Iterator>
    class move_iterator {
        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
    — random_access_iterator_tag if the type iterator_traits<Iterator>::iterator_category models derived_from<random_access_iterator_tag>, and
    — iterator_traits<Iterator>::iterator_category otherwise.

§ 23.5.3.1
23.5.3.2 Requirements

The template parameter Iterator shall either meet the Cpp17InputIterator requirements (23.3.5.2) 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 (23.3.5.5) 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 (23.3.5.6) or model random_access_iterator (23.3.4.13).

23.5.3.3 Construction and assignment

constexpr move_iterator();

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.

customexpr explicit move_iterator(Iterator i);

Effects: Constructs a move_iterator, initializing current with std::move(i).

template<class U> constexpr move_iterator(const move_iterator<U>& u);

Mandates: U is convertible to Iterator.

Effects: Constructs a move_iterator, initializing current with u.base().

template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);

Mandates: U is convertible to Iterator.

Effects: Assigns u.base() to current.

23.5.3.4 Conversion

constexpr Iterator base() const &;

Constraints: Iterator satisfies copy_constructible.

Preconditions: Iterator models copy_constructible.

Returns: current.

customexpr Iterator base() &&;

Returns: std::move(current).

23.5.3.5 Element access

constexpr reference operator*() const;

Effects: Equivalent to: return ranges::iter_move(current);

constexpr reference operator[](difference_type n) const;

Effects: Equivalent to: ranges::iter_move(current + n);

23.5.3.6 Navigation

constexpr move_iterator& operator++();

Effects: As if by ++current.

Returns: *this.

customexpr auto operator++(int);

Effects: If Iterator models forward_iterator, equivalent to:

move_iterator tmp = *this;
++current;
return tmp;

Otherwise, equivalent to ++current.
constexpr move_iterator& operator--();

Effects: As if by --current.
Returns: *this.

constexpr move_iterator operator--(int);

Effects: As if by:
move_iterator tmp = *this;
--current;
return tmp;

constexpr move_iterator operator+(difference_type n) const;

Returns: move_iterator(current + n).

constexpr move_iterator& operator+=(difference_type n);

Effects: As if by:
current += n;
Returns: *this.

constexpr move_iterator operator-(difference_type n) const;

Returns: move_iterator(current - n).

constexpr move_iterator& operator-=(difference_type n);

Effects: As if by: current -= n;
Returns: *this.

23.5.3.7 Comparisons

template<class Iterator1, class Iterator2>
constexpr bool operator==(const move_iterator<Iterator1>& x,
const move_iterator<Iterator2>& y);

template<sentinel_for<Iterator> S>
friend constexpr bool operator==(const move_iterator& x,
const move_sentinel<S>& 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 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().

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: 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().

23.5.3.8 Non-member functions

Returns: x.base() <=> y.base().

23.5.3.9 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<I, S>, move_sentinel<S> and move_iterator<I> model sentinel_for<move_sentinel<S>, move_iterator<I>> as well.

[Example: A move_if algorithm is easily implemented with copy_if using move_iterator and move_sentinel:

```cpp
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);
}
```
– end example

```cpp
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
};

23.5.3.10 Operations

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

23.5.4.1 Class template 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.

2 [Note: 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]

3 [Example:
  template<class ForwardIterator>
  void fun(ForwardIterator begin, ForwardIterator end);

  list<int> s;
  // populate the list
  using CI = common_iterator<counted_iterator<list<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<I2&, const I2&> && assignable_from<S2&, const S2&>
    common_iterator& operator=(const common_iterator<I2, S2>& x);

decltype(auto) operator*();
decltype(auto) operator*(const I&);
decltype(auto) operator->()(const I&);

template<class I2, sentinel_for<I> S2>
    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);

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>;
};

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>;
};

§ 23.5.4.1 944
23.5.4.2 Associated types

The nested typedef-names of the specialization of iterator_traits for common_iterator<, 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<, S>, then pointer denotes the type of that expression. Otherwise, pointer denotes void.

23.5.4.3 Constructors and conversions

constexpr common_iterator(I i);

Effects: Initializes v_ as if by v_{in_place_type<I>, std::move(i)}.

constexpr common_iterator(S s);

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);

Preconditions: x.v_.valueless_by_exception() is false.

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<const I2&, const I2&> && assignable_from<const S2&, const S2&>
common_iterator& operator=(const common_iterator<I2, S2>& x);

Preconditions: x.v_.valueless_by_exception() is false.

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().

Returns: *this

23.5.4.4 Accessors

dcltype(auto) operator*();

dcltype(auto) operator*() const

requires dereferenceable<const I>;

Preconditions: holds_alternative<I>(v_).

Effects: Equivalent to: return *get<I>(v_);

dcltype(auto) operator->() const

requires see below;

The expression in the requires clause is equivalent to:

indirectly_readable<const I> &&
 (requires(const I& 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_).

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_);
Otherwise, if `iter_reference_t<I>` is a reference type, equivalent to:

```cpp
auto&& tmp = *get<I>(v_);
return addressof(tmp);
```

Otherwise, equivalent to: return `proxy(*get<I>(v_));` where `proxy` is the exposition-only class:

```cpp
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

```cpp
common_iterator& operator++();
```

1. **Preconditions:** `holds_alternative<I>(v_)`.
2. **Effects:** Equivalent to `++get<I>(v_)`.
3. **Returns:** `*this`.

```cpp
decltype(auto) operator++(int);
```

1. **Preconditions:** `holds_alternative<I>(v_)`.
2. **Effects:** If `I` models `forward_iterator`, equivalent to:

```cpp
common_iterator tmp = *this;
++*this;
return tmp;
```

Otherwise, equivalent to: return `get<I>(v_)++`;

### 23.5.4.6 Comparisons

```cpp
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()`.

```cpp
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);
```

3. **Preconditions:** `x.v_.valueless_by_exception()` and `y.v_.valueless_by_exception()` are each false.
4. **Returns:** true if `i` and `j` are each 1, and otherwise `get<i>(x.v_) == get<j>(y.v_)`, where `i` is `x.v_.index()` and `j` is `y.v_.index()`.

```cpp
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);
```

5. **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 \(\text{get}\langle i \rangle(x.v_.\text{index}) - \text{get}\langle j \rangle(y.v_.\text{index})\), where \(i\) is \(x.v_.\text{index}\) and \(j\) is \(y.v_.\text{index}\).

### 23.5.4.7 Customization

```
friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
noexcept(noexcept(ranges::iter_move(declval<const I&>())))
requires input_iterator<I>;
```

1. **Preconditions:** `holds_alternative<I>(v_)`.
2. **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&>())))
requires convertible_to<const I2&, I>
requires assignable_from<I&, const I2&>;
```

3. **Preconditions:** `holds_alternative<I>(x.v_)` and `holds_alternative<I2>(y.v_)` are each true.
4. **Effects:** Equivalent to `ranges::iter_swap(get<I>(x.v_), get<I2>(y.v_));`

### 23.5.5 Default sentinels

```
namespace std {
  struct default_sentinel_t { };
}
```

1. 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`

```
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);
  };
```

3. 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.
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--();
decltype(auto) operator--(int)
  requires bidirectional_iterator<I>;
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);
friend constexpr iter_difference_t<I> operator-(
    const counted_iterator& x, default_sentinel_t);
friend constexpr iter_difference_t<I> operator-(
    default_sentinel_t, const counted_iterator& 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);
friend constexpr bool operator==(
    const counted_iterator& x, default_sentinel_t);

template<common_with<I> I2>
friend constexpr strong_ordering operator<=>(
    const counted_iterator& x, const counted_iterator<I2>& y);

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)));
23.5.6.2 Constructors and conversions

```cpp
constexpr counted_iterator(I i, iter_difference_t<I> n);
```

1. **Preconditions:** \( n \geq 0 \).
2. **Effects:** Initializes `current` with `std::move(i)` and `length` with `n`.

```cpp
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`.

```cpp
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

```cpp
constexpr I base() const & requires copy_constructible<I>;
```

1. **Effects:** Equivalent to: `return current;`

```cpp
constexpr I base() &&;
```

2. **Returns:** `std::move(current)`.

```cpp
constexpr iter_difference_t<I> count() const noexcept;
```

3. **Effects:** Equivalent to: `return length;`

23.5.6.4 Element access

```cpp
constexpr decltype(auto) operator*();
```

1. **Effects:** Equivalent to: `return *current;`

```cpp
constexpr decltype(auto) operator*() const
requires dereferenceable<const I>;
```

2. **Preconditions:** \( n < length \).
3. **Effects:** Equivalent to: `return current[n];`

23.5.6.5 Navigation

```cpp
constexpr counted_iterator& operator++();
```

1. **Preconditions:** `length > 0`.
2. **Effects:** Equivalent to:
   ```cpp
   ++current;
   --length;
   return *this;
   ```
decltype(auto) operator++(int);

**Preconditions:** length > 0.

**Effects:** Equivalent to:

```cpp
--length;
try { return current++; }
catch(...) { ++length; throw; }
```

```cpp
constexpr counted_iterator operator++(int)
requires forward_iterator<I>;
```

**Effects:** Equivalent to:

```cpp
counted_iterator tmp = *this;
++*this;
return tmp;
```

```cpp
constexpr counted_iterator& operator--()
requires bidirectional_iterator<I>;
```

**Effects:** Equivalent to:

```cpp
--current;
++length;
return *this;
```

```cpp
constexpr counted_iterator operator--(int)
requires bidirectional_iterator<I>;
```

**Effects:** Equivalent to:

```cpp
counted_iterator tmp = *this;
--*this;
return tmp;
```

```cpp
constexpr counted_iterator operator+(iter_difference_t<I> n) const
requires random_access_iterator<I>;
```

**Effects:** Equivalent to:

```cpp
return counted_iterator(current + n, length - n);
```

```cpp
friend constexpr counted_iterator operator+(iter_difference_t<I> n, const counted_iterator& x)
requires random_access_iterator<I>;
```

**Effects:** Equivalent to: return x + n;

```cpp
constexpr counted_iterator& operator+=(iter_difference_t<I> n)
requires random_access_iterator<I>;
```

**Preconditions:** n <= length.

**Effects:** Equivalent to:

```cpp
current += n;
length -= n;
return *this;
```

```cpp
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);

```cpp
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;

```cpp
friend constexpr iter_difference_t<I> operator-(
    const counted_iterator& x, default_sentinel_t);
```

**Effects:** Equivalent to: return -x.length;

§ 23.5.6.5
friend constexpr iter_difference_t<I> operator-(
   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: 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)
noexcept(noexcept(ranges::iter_move(i.current)))
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)
noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

Effects: Equivalent to ranges::iter_swap(x.current, y.current).

23.5.7 Unreachable sentinel

[unreachable.sentinels]

23.5.7.1 Class unreachable_sentinel_t

[unreachable.sentinel]

Class unreachable_sentinel_t can be used with any weakly_incrementable type to denote the “upper bound” of an unbounded interval.

[Example:

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]

§ 23.5.7.1
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

To make it possible for algorithmic templates to work directly with input/output streams, appropriate
iterator-like class templates are provided.

[Example:
    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

23.6.1 Class template istream_iterator

The class template istream_iterator is an input iterator (23.3.5.2) 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&);
            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
        };
    }
}

The type T shall meet the Cpp17DefaultConstructible, Cpp17CopyConstructible, and Cpp17CopyAssign able
requirements.

23.6.1.1 Constructors and destructor

constexpr istream_iterator();
constexpr istream_iterator(default_sentinel_t);

**Effects:** Constructs the end-of-stream iterator, value-initializing value.

**Postconditions:** in_stream == nullptr is true.

**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);

**Effects:** Initializes in_stream with addressof(s), value-initializes value, and then calls operator++().

istream_iterator(const istream_iterator& x) = default;

**Postconditions:** in_stream == x.in_stream is true.

**Remarks:** If is_trivially_copy_constructible_v<T> is true, then this constructor is trivial.

~istream_iterator() = default;

**Remarks:** If is_trivially_destructible_v<T> is true, then this destructor is trivial.

### 23.6.1.2 Operations

```
const T& operator*() const;

Preconditions: in_stream != nullptr is true.

Returns: value.
```

```
const T* operator->() const;

Preconditions: in_stream != nullptr is true.

Returns: addressof(value).
```

```
istream_iterator& operator++();

Preconditions: in_stream != nullptr is true.

Effects: Equivalent to:

```
if (!(*in_stream >> value))
   in_stream = nullptr;
```

Returns: *this.
```

```
istream_iterator operator++(int);

Preconditions: in_stream != nullptr is true.

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);
```

**Returns:** x.in_stream == y.in_stream.

```
friend bool operator==(const istream_iterator& i, default_sentinel_t);
```

**Returns:** !i.in_stream.

### 23.6.2 Class template ostream_iterator

**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.

§ 23.6.2 953
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
  };
}

23.6.2.1 Constructors and destructor

ostream_iterator(ostream_type& s);

1 Effects: Initializes out_stream with addressof(s) and delim with nullptr.

ostream_iterator(ostream_type& s, const charT* delimiter);

2 Effects: Initializes out_stream with addressof(s) and delim with delimiter.

23.6.2.2 Operations

ostream_iterator& operator=(const T& value);

1 Effects: As if by:
   *out_stream << value;
   if (delim)
      *out_stream << delim;
   return *this;

ostream_iterator& operator*();

2 Returns: *this.

ostream_iterator& operator++();

ostream_iterator& operator++(int);

3 Returns: *this.

23.6.3 Class template istreambuf_iterator

The class template istreambuf_iterator defines an input iterator (23.3.5.2) 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

§ 23.6.3 954
end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

2 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
        }
    }
}
```

### 23.6.3.1 Class istreambuf_iterator::proxy

1 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.

```cpp
namespace std {
    template<class charT, class traits>
    class istreambuf_iterator<charT, traits>::proxy { // exposition only
        charT keep_;  
        basic_streambuf<charT,traits>* sbuf_;  
        proxy(charT c, basic_streambuf<charT,traits>* sbuf) : keep_(c), sbuf_(sbuf) { }

        public:
            charT operator*() { return keep_; }
        }
    }
}
```
23.6.3.2 Constructors

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.

```cpp
constexpr istreambuf_iterator() noexcept;

effects: Initializes `sbuf_` with `nullptr`.

constexpr istreambuf_iterator(default_sentinel_t) noexcept;

effects: Initializes `sbuf_` with `s.rdbuf()`.

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_`.
```

23.6.3.3 Operations

```cpp
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.
```

```cpp
template<class charT, class traits>
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)`.
```

23.6.4 Class template ostreambuf_iterator

The class template `ostreambuf_iterator` writes successive `characters` onto the output stream from which it was constructed.

```cpp
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
};

23.6.4.1 Constructors

ostreambuf_iterator(ostream_type& s) noexcept;
  
  **Preconditions:** s.rdbuf() is not a null pointer.

  **Effects:** Initializes sbuf_ with s.rdbuf().

ostreambuf_iterator(streambuf_type* s) noexcept;
  
  **Preconditions:** s is not a null pointer.

  **Effects:** Initializes sbuf_ with s.

23.6.4.2 Operations

ostreambuf_iterator& operator=(charT c);
  
  **Effects:** If failed() yields false, calls sbuf_->sputc(c); otherwise has no effect.

  **Returns:** *this.

ostreambuf_iterator& operator*();
  
  **Returns:** *this.

ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);
  
  **Returns:** *this.

bool failed() const noexcept;
  
  **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> (30.4), <set> (22.4.3), <span> (22.7.2),<string> (21.3.1),
<string_view> (21.4.1), <unordered_map> (22.5.2), <unordered_set> (22.5.3), and <vector> (22.3.6). Each
of these templates is a designated customization point (16.5.4.2.1).

template<class C> constexpr auto begin(C& c) noexcept = default;
template<class C> constexpr auto begin(const C& c) noexcept = default;

template<class C> constexpr auto end(C& c) noexcept = default;
template<class C> constexpr auto end(const C& c) noexcept = default;

template<
class T, size_t N
> constexpr T* begin(T (&array)[N]) noexcept;
  
  **Returns:** array.
template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;

Returns: array + N.

template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
-> decltype(std::begin(c));

Returns: std::begin(c).

template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));

Returns: 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());

Returns: c.rbegin().

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().

template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);

Returns: reverse_iterator<T*>(array + N).

template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);

Returns: reverse_iterator<T*>(array).

template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);

Returns: reverse_iterator<const E*>(il.end()).

template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);

Returns: reverse_iterator<const E*>(il.begin()).

template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));

template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));

Returns: std::rbegin(c).

Returns: std::rend(c).

template<class C> constexpr auto size(const C& c) -> decltype(c.size());

Returns: c.size().

template<class T, size_t N> constexpr size_t size(const T (&array)[N]) noexcept;

Returns: N.

template<class C> constexpr auto ssize(const C& c)
-> 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 Range access</td>
<td>&lt;ranges&gt;</td>
</tr>
<tr>
<td>24.4 Requirements</td>
<td></td>
</tr>
<tr>
<td>24.5 Range utilities</td>
<td></td>
</tr>
<tr>
<td>24.6 Range factories</td>
<td></td>
</tr>
<tr>
<td>24.7 Range adaptors</td>
<td></td>
</tr>
</tbody>
</table>

24.2 Header <ranges> synopsis

```cpp
#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.1
#include <iterator>  // see 23.2

namespace std::ranges {

  // 24.3, range access
  inline namespace unspecified {
    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<iterator_t<R>>;```
template<class sized_range R>
    using range_size_t_t = decltype(ranges::size(declval<R&>()));
template<class range R>
    using range_value_t = iter_value_t<iterator_t<R>>;
template<class range R>
    using range_reference_t = iter_reference_t<iterator_t<R>>;
template<class range R>
    using range_rvalue_reference_t = iter_rvalue_reference_t<iterator_t<R>>;

// 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;

// 24.5.2, class template view_interface
template<class D>
    requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface;

// 24.5.3, 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;
// 24.5.4, 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>;

// 24.6.1, 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{};
}

// 24.6.2, single view
template<copy_constructible T>
requires is_object_v<T>
class single_view;

namespace views { inline constexpr unspecified single = unspecified; }

// 24.6.3, iota view
template<weakly_incrementable W, semiregular Bound = unreachable_sentinel_t>
requires weakly-equality-comparable-with<W, Bound>
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; }

// 24.6.4, 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);

// 24.7.3, all view
namespace views {
    inline constexpr unspecified all = unspecified;

    template<viewable_range R>
    using all_t = decrtype(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;
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; }

template<input_range V, copy_constructible F>
    requires view<V> && is_object_v<F> &&
    regular_invocable<F&, range_reference_t<V>>
    class transform_view;

namespace views { inline constexpr unspecified transform = unspecified; }

template<view> class take_view;

namespace views { inline constexpr unspecified take = unspecified; }

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; }

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; }

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; }
// 24.7.12, counted view
namespace views { inline constexpr unspecified counted = unspecified; }

// 24.7.13, common view
template<view V>
    requires (!common_range<V> && copyable<iterator_t<V>>)
class common_view;
namespace views { inline constexpr unspecified common = unspecified; }

// 24.7.14, reverse view
template<view V>
    requires bidirectional_range<V>
class reverse_view;
namespace views { inline constexpr unspecified reverse = unspecified; }

// 24.7.15, 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

1 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.

2 Within this subclause, the *reified object* of a subexpression E denotes

(2.1) — the same object as E if E is a glvalue, or

(2.2) — the result of applying the temporary materialization conversion (7.3.4) to E otherwise.

24.3.1 ranges::begin

1 The name `ranges::begin` denotes a customization point object (16.4.2.2.6).

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 `enable_borrowed_range<remove_cv_t<T>>` is false, `ranges::begin(E)` is ill-formed.

(2.2) — Otherwise, if T is an array type (6.8.2) and `remove_all_extents_t<T>` is an incomplete type, `ranges::begin(E)` is ill-formed with no diagnostic required.

(2.3) — Otherwise, if T is an array type, `ranges::begin(E)` is expression-equivalent to t + 0.

(2.4) — 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())`.

(2.5) — Otherwise, if T is a class or enumeration type and `decay-copy(end(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

```cpp
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.

(2.6) — Otherwise, `ranges::begin(E)` is ill-formed.

3 [Note: Diagnosable ill-formed cases above result in substitution failure when `ranges::begin(E)` appears in the immediate context of a template instantiation. — end note]

4 [Note: Whenever `ranges::begin(E)` is a valid expression, its type models `input_or_output_iterator`. — end note]

24.3.2 ranges::end

1 The name `ranges::end` denotes a customization point object (16.4.2.2.6).

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 `enable_borrowed_range<remove_cv_t<T>>` is false, `ranges::end(E)` is ill-formed.

(2.2) — Otherwise, if T is an array type (6.8.2) and `remove_all_extents_t<T>` is an incomplete type, `ranges::end(E)` is ill-formed with no diagnostic required.

(2.3) — Otherwise, if T is an array of unknown bound, `ranges::end(E)` is ill-formed.

(2.4) — Otherwise, if T is an array, `ranges::end(E)` is expression-equivalent to t + extent_v<T>.

(2.5) — 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())`.

(2.6) — 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

```cpp
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.

(2.7) — Otherwise, `ranges::end(E)` is ill-formed.
24.3.3 ranges::cbegin

The name ranges::cbegin denotes a customization point object (16.4.2.2.6). The expression ranges::cbegin(E) for a subexpression E of type T is expression-equivalent to:

1. (1.1) ranges::begin(static_cast<const T&>(E)) if E is an lvalue.
2. (1.2) Otherwise, ranges::begin(static_cast<const T&&>(E)).

2 [Note: Whenever ranges::cbegin(E) is a valid expression, its type models input_or_output_iterator. —end note]

24.3.4 ranges::cend

The name ranges::cend denotes a customization point object (16.4.2.2.6). The expression ranges::cend(E) for a subexpression E of type T is expression-equivalent to:

1. (1.1) ranges::end(static_cast<const T&>(E)) if E is an lvalue.
2. (1.2) Otherwise, ranges::end(static_cast<const T&&>(E)).

2 [Note: Whenever ranges::cend(E) is a valid expression, the types S and I of ranges::cend(E) and ranges::cbegin(E) model sentinel_for<S, I>. —end note]

24.3.5 ranges::rbegin

The name ranges::rbegin denotes a customization point object (16.4.2.2.6).

2 Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:

1. (2.1) If E is an rvalue and enable_borrowed_range<remove_cv_t<T>> is false, ranges::rbegin(E) is ill-formed.
2. (2.2) Otherwise, if T is an array type (6.8.2) and remove_all_extents_t<T> is an incomplete type, ranges::rbegin(E) is ill-formed with no diagnostic required.
3. (2.3) Otherwise, if decay-copy(t.rbegin()) is a valid expression whose type models input_or_output_iterator, ranges::rbegin(E) is expression-equivalent to decay-copy(t.rbegin()).
4. (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.
5. (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)).
6. (2.6) Otherwise, ranges::rbegin(E) is ill-formed.

3 [Note: Diagnosable ill-formed cases above result in substitution failure when ranges::rbegin(E) appears in the immediate context of a template instantiation. —end note]

4 [Note: Whenever ranges::rbegin(E) is a valid expression, its type models input_or_output_iterator. —end note]

24.3.6 ranges::rend

The name ranges::rend denotes a customization point object (16.4.2.2.6).

2 Given a subexpression E with type T, let t be an lvalue that denotes the reified object for E. Then:

1. (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 (6.8.2) and `remove_all_extents_t<T>` is an incomplete type, `ranges::rend(E)` is ill-formed with no diagnostic required.

— Otherwise, if `decay-copy(t.rend())` is a valid expression whose type models `sentinel_for<decl-type(ranges::rbegin(E))>` then `ranges::rend(E)` is expression-equivalent to `decay-copy(t.rend())`.

— Otherwise, if $T$ is a class or enumeration type and `decay-copy(rbegin(t))` is a valid expression whose type models `sentinel_for<decl-type(ranges::rbegin(E))>` with overload resolution performed in a context in which unqualified lookup for `rend` finds only the declarations

```cpp
void rend(auto&) = delete;
void rend(const auto&) = delete;
```

then `ranges::rend(E)` is expression-equivalent to `decay-copy(rend(t))` with overload resolution performed in the above context.

— 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), then `ranges::rend(E)` is expression-equivalent to `make_reverse_iterator(ranges::begin(t))`.

— Otherwise, `ranges::rend(E)` is ill-formed.

3 [Note: Diagnosable ill-formed cases above result in substitution failure when `ranges::rend(E)` appears in the immediate context of a template instantiation. —end note]

4 [Note: Whenever `ranges::rend(E)` is a valid expression, the types $S$ and $I$ of `ranges::rend(E)` and `ranges::rbegin(E)` model `sentinel_for<S, I>`. —end note]

### 24.3.7 `ranges::crebegin`

The name `ranges::crebegin` denotes a customization point object (16.4.2.2.6). The expression `ranges::crebegin(E)` for a subexpression $E$ of type $T$ is expression-equivalent to:

1. **(1.1)** `ranges::rbegin(static_cast<const T&>(E))` if $E$ is an lvalue.
2. **(1.2)** Otherwise, `ranges::rbegin(static_cast<const T&&>(E))`.

2 [Note: Whenever `ranges::crebegin(E)` is a valid expression, its type models `input_or_output_iterator`. —end note]

### 24.3.8 `ranges::crend`

The name `ranges::crend` denotes a customization point object (16.4.2.2.6). The expression `ranges::crend(E)` for a subexpression $E$ of type $T$ is expression-equivalent to:

1. **(1.1)** `ranges::rend(static_cast<const T&>(E))` if $E$ is an lvalue.
2. **(1.2)** Otherwise, `ranges::rend(static_cast<const T&&>(E))`.

2 [Note: Whenever `ranges::crend(E)` is a valid expression, the types $S$ and $I$ of `ranges::crend(E)` and `ranges::crbegin(E)` model `sentinel_for<S, I>`]. —end note]

### 24.3.9 `ranges::size`

The name `ranges::size` denotes a customization point object (16.4.2.2.6).

2 Given a subexpression $E$ with type $T$, let $t$ be an lvalue that denotes the reified object for $E$. Then:

1. **(2.1)** If $T$ is an array of unknown bound (9.3.3.4), `ranges::size(E)` is ill-formed.
2. **(2.2)** Otherwise, if $T$ is an array type, `ranges::size(E)` is expression-equivalent to `decay-copy(extent_v<T>)`.
3. **(2.3)** Otherwise, if `disable_sized_range<remove_cv_t<T>>(24.4.3)` is `false` and `decay-copy(t.size())` is a valid expression of integer-like type (23.3.4.4), `ranges::size(E)` is expression-equivalent to `decay-copy(t.size())`.
4. **(2.4)** Otherwise, if $T$ is a class or enumeration type, `disable_sized_range<remove_cv_t<T>>(24.4.3)` is `false` and `decay-copy(size(t))` is a valid expression of integer-like type with overload resolution performed in a context in which unqualified lookup for `size` finds only the declarations

```cpp
void size(auto&) = delete;
void size(const auto&) = delete;
```
then `ranges::size(E)` is expression-equivalent to \textit{decay-copy(size(t))} with overload resolution performed in the above context.

(2.5) Otherwise, if \textit{to-unsigned-like(ranges::end(t) - ranges::begin(t))} (24.2) is a valid expression and the types I and S of \textit{ranges::begin(t)} and \textit{ranges::end(t)} (respectively) model both \textit{sized_sentinel_for<S, I>} (23.3.4.8) and \textit{forward_iterator<I>}, then `ranges::size(E)` is expression-equivalent to \textit{to-unsigned-like(ranges::end(t) - ranges::begin(t))}.

(2.6) Otherwise, `ranges::size(E)` is ill-formed.

3 [Note: Diagnosable ill-formed cases above result in substitution failure when `ranges::size(E)` appears in the immediate context of a template instantiation. —end note]

4 [Note: Whenever `ranges::size(E)` is a valid expression, its type is integer-like. —end note]

24.3.10 \texttt{ranges::ssize}\[range.prim.ssize\]
1 The name \texttt{ranges::ssize} denotes a customization point object (16.4.2.2.6). The expression \texttt{ranges::ssize(E)} for a subexpression E of type T is expression-equivalent to:

(1.1) If \texttt{range_difference_t<T> has width less than ptrdiff_t}, \texttt{static_cast<ptrdiff_t>(ranges::size(E))}.

(1.2) Otherwise, \texttt{static_cast<range_difference_t<T>>(ranges::size(E))}.

24.3.11 \texttt{ranges::empty}\[range.prim.empty\]
1 The name \texttt{ranges::empty} denotes a customization point object (16.4.2.2.6).

2 Given a subexpression \texttt{ranges::empty(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 (6.8.2), \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 \textit{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.

3 [Note: Diagnosable ill-formed cases above result in substitution failure when \texttt{ranges::empty(E)} appears in the immediate context of a template instantiation. —end note]

4 [Note: Whenever \texttt{ranges::empty(E)} is a valid expression, it has type \texttt{bool}. —end note]

24.3.12 \texttt{ranges::data}\[range.prim.data\]
1 The name \texttt{ranges::data} denotes a customization point object (16.4.2.2.6).

2 Given a subexpression \texttt{ranges::data(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<T>}} is false, \texttt{ranges::data(E)} is ill-formed.

(2.2) Otherwise, if T is an array type (6.8.2) and \texttt{remove_all_extents_t<T>}} 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 \textit{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.

3 [Note: Diagnosable ill-formed cases above result in substitution failure when \texttt{ranges::data(E)} appears in the immediate context of a template instantiation. —end note]

4 [Note: Whenever \texttt{ranges::data(E)} is a valid expression, it has pointer to object type. —end note]
24.3.13  ranges::cdata

The name ranges::cdata denotes a customization point object (16.4.2.2.6). The expression ranges::cdata(E) for a subexpression E of type T is expression-equivalent to:

(1.1) ranges::cdata(static_cast<const T&>(E)) if E is an lvalue.

(1.2) Otherwise, ranges::cdata(static_cast<const T&&>(E)).

[Note: 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.

```
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).

1 Given an expression t such that decltype((t)) is T& T models range only if

(3.1) [ranges::begin(t), ranges::end(t)] denotes a range (23.3.1),

(3.2) both ranges::begin(t) and ranges::end(t) are amortized constant time and non-modifying, and

(3.3) if the type of ranges::begin(t) models forward_iterator, ranges::begin(t) is equality-preserving.

[Note: 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; ranges::begin should be called at most once for such a range. — end note]

```
template<class T>
concept borrowed_range =
  range<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: 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]

```
template<class T>
inline constexpr bool enable_borrowed_range = false;
```

Remarks: Pursuant to 16.5.4.2.1, 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: Each specialization $S$ of class template `subrange` (24.5.3) models `borrowed_range` because

(8.1) $S$’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`.

```
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

(2.1) `ranges::size(t)` is amortized $\Theta(1)$, does not modify $t$, and is equal to `ranges::distance(t)`, and

(2.2) if `iterator_t<T>` models `forward_iterator`, `ranges::size(t)` is well-defined regardless of the evaluation of `ranges::begin(t)`. [Note: `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]

### 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`.

```
template<class T>
concept view =
    range<T> && movable<T> && default_initializable<T> && enable_view<T>;
```

$T$ models `view` only if:

(2.1) $T$ has $\Theta(1)$ move construction; and

(2.2) $T$ has $\Theta(1)$ move assignment; and

(2.3) $T$ has $\Theta(1)$ destruction; and
copy_constructible<T> is false, or T has $O(1)$ copy construction; and

— copyable<T> is false, or T has $O(1)$ copy assignment.

[Example: 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. — end example]

Since the difference between range and view is largely semantic, the two are differentiated with the help of enable_view.

```cpp
template<class T>
inline constexpr bool enable_view = derived_from<T, view_base>;
```

Remarks: Pursuant to 16.5.4.2.1, users may specialize enable_view to true for cv-qualified 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.

# Other range refinements [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.

```cpp
template<class R, class T>
concept output_range =
  range<R> && output_iterator<iterator_t<R>, T>;
```

```cpp
template<class T>
concept input_range =
  range<T> && input_iterator<iterator_t<T>>;
```

```cpp
template<class T>
concept forward_range =
  input_range<T> && forward_iterator<iterator_t<T>>;
```

```cpp
template<class T>
concept bidirectional_range =
  forward_range<T> && bidirectional_iterator<iterator_t<T>>;
```

```cpp
template<class T>
concept random_access_range =
  bidirectional_range<T> && random_access_iterator<iterator_t<T>>;
```

contiguous_range additionally requires that the ranges::data customization point (24.3.12) is usable with the range.

```cpp
template<class T>
concept contiguous_range =
  random_access_range<T> && contiguous_iterator<iterator_t<T>>
  && requires(T& t) {
    { ranges::data(t) } -> same_as<add_pointer_t<range_reference_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: The standard containers (Clause 22) model common_range.

— end example]
template<class T>
    concept common_range =
        range<T> && same_as<iterator_t<T>, sentinel_t<T>>;

The \texttt{viewable\_range} concept specifies the requirements of a \texttt{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>>);

\section{Range utilities} \label{range.utility}
\index{utility!range}

The components in this subclause are general utilities for representing and manipulating ranges.

\subsection{Helper concepts} \label{range.utility.helpers}
Many of the types in subclause \ref{range.utility} are specified in terms of the following exposition-only concepts:

\begin{verbatim}
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>>;
\end{verbatim}

\subsection{View interface} \label{view.interface}
The class template \texttt{view\_interface} is a helper for defining \texttt{view}-like types that offer a container-like interface. It is parameterized with the type that is derived from it.

\begin{verbatim}
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());
                    }
\end{verbatim}

\section*{§ 24.5.2}
2 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`.

### 24.5.2.1 Members

```cpp
constexpr decltype(auto) front() requires forward_range<D>;
constexpr decltype(auto) front() const requires forward_range<const D>;

constexpr decltype(auto) back() requires bidirectional_range<D> && common_range<D>;
constexpr decltype(auto) back() const
requires bidirectional_range<const D> && common_range<const D>;
```

1 Preconditions: `!empty()`.

2 Effects: Equivalent to: `return *ranges::begin(derived());`

```cpp
template<random_access_range R = D>
constexpr decltype(auto) operator[](range_difference_t<R> n) {
    return ranges::begin(derived())[n];
}
```

```cpp
namespace std::ranges {
    template<class From, class To>
    concept convertible_to_non_slicing =
        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>>>);

```cpp
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>&>;
};
```

```cpp
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>> &&
  convertible_to<V, tuple_element_t<1, T>>;
```

```cpp
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>>;
```

```cpp
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(// exposition only
    convertible-to-non-slicing<I> auto i, S s) requires (!StoreSize);

  constexpr subrange(// exposition only
    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 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>;  

namespace std {  
using ranges::get;  
}

24.5.3.1 Constructors and conversions

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)).

§ 24.5.3.1
Effects: Initializes `begin_` with `std::move(i)` and `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `n`.

[Note: Accepting the length of the range and storing it to later return from `size()` enables `subrange` to model `sized_range` even when it stores an iterator and sentinel that do not model `sized_sentinel_for`. — end note]

```cpp
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>);
```

Effects: Equivalent to:

(6.1) If `StoreSize` is `true`, `subrange{r, ranges::size(r)}`.
(6.2) Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

```cpp
template<not_same_as<subrange> PairLike>
requires pair_like_convertible_from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
```

Effects: Equivalent to: return `PairLike(begin_, end_)`;

### 24.5.3.2 Accessors

```cpp
constexpr I begin() const requires copyable<I>;
```

Effects: Equivalent to: return `begin_`;

```
[[nodiscard]] constexpr I begin() requires (!copyable<I>);
```

Effects: Equivalent to: return `std::move(begin_)`;

```cpp
constexpr S end() const;
```

Effects: Equivalent to: return `end_`;

```cpp
constexpr bool empty() const;
```

Effects: Equivalent to: return `begin_ == end_`;

```cpp
constexpr make_unsigned_like<iter_difference_t<I>> size() const
requires (K == subrange_kind::sized);
```

Effects:

(5.1) If `StoreSize` is `true`, equivalent to: return `size_`;
(5.2) Otherwise, equivalent to: return `to_unsigned_like(end_ - begin_)`;

```cpp
[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const &
requires forward_iterator<I>;
```

Effects: Equivalent to:

```cpp
auto tmp = *this;
tmp.advance(n);
return tmp;
```

```cpp
[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) &&;
```

Effects: Equivalent to:

```cpp
advance(n);
return std::move(*this);
```

```cpp
[[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const
requires bidirectional_iterator<I>;
```

Effects: Equivalent to:

```cpp
auto tmp = *this;
tmp.advance(-n);
return tmp;
```
constexpr subrange& advance(iter_difference_t<I> n);

Effects: Equivalent to:

(9.1) 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;
```

(9.2) Otherwise,

```cpp
ranges::advance(begin_, n, end_);
return *this;
```

template<size_t N, class I, class S, subrange_kind K> requires (N < 2)
constexpr auto get(const subrange<I, S, K>& r);

24.5.4 Dangling iterator handling [range.dangling]

The tag type dangling is used together with the template aliases borrowed_iterator_t and borrowed_subrange_t to indicate that an algorithm that typically returns an iterator into or subrange of a range argument does not return an iterator or subrange which could potentially reference a range whose lifetime has ended for a particular rvalue range argument which does not model borrowed_range (24.4.2).

```cpp
namespace std::ranges {
    struct dangling {
        constexpr dangling() noexcept = default;
        template<class... Args> constexpr dangling(Args&&...) noexcept { }
    };
}
```

```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.

---

24.6 Range factories [range.factories]

This subclause defines range factories, which are utilities to create a view.

Range factories are declared in namespace std::ranges::views.
24.6.1 Empty view

24.6.1.1 Overview

`empty_view` produces a `view` of no elements of a particular type.

1 [Example:
   
   ```cpp
   empty_view<int> e;
   static_assert(ranges::empty(e));
   static_assert(0 == e.size());
   -- end example]

24.6.1.2 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; }
    };
}

24.6.2 Single view

24.6.2.1 Overview

`single_view` produces a `view` that contains exactly one element of a specified value.

1 The name `views::single` denotes a customization point object (16.4.2.2.6). Given a subexpression `E`, the expression `views::single(E)` is expression-equivalent to `single_view{E}`.

2 [Example:
   
   ```cpp
   single_view s{4};
   for (int i : s)
       cout << i; // prints 4
   -- end example]

24.6.2.2 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 24.7.2)
    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;
    };
}

§ 24.6.2.2
constexpr explicit single_view(const T& t);

Effects: Initializes value_ with t.

constexpr explicit single_view(T&& t);

Effects: Initializes value_ with std::move(t).

template<class... 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;

Effects: Equivalent to: return 1;

constexpr T* data() noexcept;
constexpr const T* data() const noexcept;

Effects: Equivalent to: return value_.operator->();

24.6.3 Iota view

24.6.3.1 Overview

iota_view generates a sequence of elements by repeatedly incrementing an initial value.

The name views::iota denotes a customization point object (16.4.2.2.6). 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:

for (int i : iota_view{1, 10})
    cout << i << ' '; // prints: 1 2 3 4 5 6 7 8 9

—end example]

24.6.3.2 Class template 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:
            // 24.6.3.3, class iota_view::iterator
            struct iterator;
                // exposition only
            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;
};

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>;

Let \textit{IOTA-DIFF-T}(W) be defined as follows:

(1.1) — If \textit{W} is not an integral type, or if it is an integral type and sizeof(iter_difference_t\langle W\rangle) is greater than sizeof(W), then \textit{IOTA-DIFF-T}(W) denotes iter_difference_t\langle W\rangle.

(1.2) — Otherwise, \textit{IOTA-DIFF-T}(W) is a signed integer type of width greater than the width of \textit{W} if such a type exists.

(1.3) — Otherwise, \textit{IOTA-DIFF-T}(W) is an unspecified signed-integer-like type (23.3.4.4) of width not less than the width of \textit{W}. [Note: It is unspecified whether this type satisfies weakly_incrementable. — end note]

The exposition-only \textbf{decrementable} concept is equivalent to:

\begin{verbatim}
    template<class I>
    concept decrementable =
        incrementable<I> && requires(I i) {
            { --i } -> same_as<I&>;
            { i-- } -> same_as<I>;
        };
\end{verbatim}

When an object is in the domain of both pre- and post-decrement, the object is said to be \textbf{decrementable}.

Let \textit{a} and \textit{b} be equal objects of type \textit{I}. \textit{I} models \textbf{decrementable} only if

(4.1) — \textit{a} and \textit{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 \textit{a} and \textit{b} are incrementable, then bool(--(++a) == b).

The exposition-only \textbf{advanceable} concept is equivalent to:

\begin{verbatim}
    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)>;
        };
\end{verbatim}

Let \textit{D} be \textit{IOTA-DIFF-T}(I). Let \textit{a} and \textit{b} be objects of type \textit{I} such that \textit{b} is reachable from \textit{a} after \textit{n} applications of ++\textit{a}, for some value \textit{n} of type \textit{D}. \textit{I} models \textbf{advanceable} only if

(5.1) — (a += n) is equal to \textit{b}.
— 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.
(5.6) — If I(a + D(n - 1)) is well-defined, then I(a + n) is equal to [I(I c) { return ++c; })(I(a + D(n - 1)))].
(5.7) — (b += -n) is equal to a.
(5.8) — (b -= n) is equal to a.
(5.9) — addressof(b -= n) is equal to addressof(b).
(5.10) — I(b - n) is equal to (b -= n).
(5.11) — D(b - a) is equal to n.
(5.12) — D(a - b) is equal to D(-n).
(5.13) — bool(a <= b) is true.

constexpr explicit iota_view(W value);
6 Preconditions: Bound denotes unreachable_sentinel_t or Bound() is reachable from value.
7 Effects: Initializes value_ with value.

custom iota_view(type_identity_t<W> value, type_identity_t<Bound> bound);
8 Preconditions: Bound denotes unreachable_sentinel_t or bound is reachable from value. When W and Bound model totally_ordered_with, then bool(value <= bound) is true.
9 Effects: Initializes value_ with value and bound_ with bound.

custom iterator begin() const;
10 Effects: Equivalent to: return iterator{value_};

custom auto end() const;
11 Effects: Equivalent to:
   if constexpr (same_as<Bound, unreachable_sentinel_t>)
      return unreachable_sentinel;
   else
      return sentinel{bound_};

custom iterator end() const requires same_as<W, Bound>;
12 Effects: Equivalent to: return iterator{bound_};

custom auto size() const requires see below;
13 Effects: Equivalent to:
   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_);
14 Remarks: The expression in the requires-clause is equivalent to
   (same_as<W, Bound> && advanceable<W>) || (integral<W> && integral<Bound>) ||
   sized_sentinel_for<Bound, W> |

24.6.3.3 Class iota_view::iterator
24.6.3.3.3 Class iota_view::iterator
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;
    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: 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_;`

   [Note: 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:  
   ```
   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--(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;
   ```

§ 24.6.3.3
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_;

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_;
}

24.6.3.4 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 {

§ 24.6.3.4 984
private:
    Bound bound_ = Bound();  // exposition only
public:
    sentinel() = default;
    constexpr explicit sentinel(Bound bound);

friend constexpr bool operator==(const iterator& x, const sentinel& y);
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);

24.6.4 Istream view

24.6.4.1 Overview

basic_istream_view models input_range and reads (using operator>>) successive elements from its corresponding input stream.

[Example:
    auto ints = istreamstringstream("0 1 2 3 4");
    ranges::copy(basic_istream_view<int>(ints), ostream_iterator<int>{cout, "-"});
    // prints 0-1-2-3-4-
    — end example]

24.6.4.2 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_;
                }
            }

§ 24.6.4.2
constexpr default_sentinel_t end() const noexcept;

private:
struct iterator; // exposition only
basic_istream<CharT, Traits>* stream_ = nullptr; // exposition only
Val object_ = Val(); // exposition only
};

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};

24.6.4.3 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 {
    public:
        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&) = delete;
        iterator& operator=(iterator&&) = default;

        iterator& operator++();
        void operator++(int);

        Val& operator*() const;

        friend bool operator==(const iterator& x, default_sentinel_t);

    private:
        basic_istream_view* parent_ = nullptr; // exposition only
    }
}

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_;

§ 24.6.4.3
24.7 Range adaptors

This subclause 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:

```cpp
type int e; 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)) {
    cout << i << ' '; // prints: 0416
} assert(ranges::equal(ints | views::filter(even), views::filter(ints, even));
```
—end example]

24.7.1 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:

```cpp
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:

```cpp
R | C | D
R | (C | D)
```

A range adaptor object is a customization point object (16.4.2.2.6) 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:

```cpp
adaptor(range, args...)
adaptor(args...)(range)
range | adaptor(args...)
```

In this case, adaptor(args...) is a range adaptor closure object.

24.7.2 Semiregular wrapper

Many types in this subclause are specified in terms of an exposition-only class template semiregular-box. 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>.
If \( T \) models `default_initializable`, the default constructor of `semiregular-box<T>` is equivalent to:

```cpp
cconstexpr semiregular-box() noexcept(is_nothrow_default_constructible_v<T>)
  : semiregular-box{in_place}
{
}
```

If `assignable_from<T&, const T&>` is not modeled, the copy assignment operator is equivalent to:

```cpp
csemiregular-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
csemiregular-box& operator=(semiregular-box&& that)
  noexcept(is_nothrow_move_constructible_v<T>)
{
  if (that) emplace(std::move(*that));
  else reset();
  return *this;
}
```

## 24.7.3 All view

1. `views::all` returns a view that includes all elements of its `range` argument.

2. The name `views::all` denotes a range adaptor object (24.7.1). Given a subexpression `E`, the expression `views::all(E)` is expression-equivalent to:

   - `decay-copy(E)` if the decayed type of `E` models `view`.
   - Otherwise, `ref_view(E)` if that expression is well-formed.
   - Otherwise, `subrange(E)`.

### 24.7.3.1 Class template `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>
      { return ranges::size(*r_); }
  }
} // namespace std::ranges
```

§ 24.7.3.1 988
constexpr auto data() const requires contiguous_range<
{ return ranges::data(*r_); }
};

template<class R>
ref_view(R&) -> ref_view<R>;

template<not_same_as<ref_view> T>
requires see below
constexpr ref_view(T&& t);

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>()); }

Effects: Initializes r_ with addressof(static_cast<R&>(std::forward<T>(t))).

24.7.4 Filter view

24.7.4.1 Overview

filter_view presents a view of the elements of an underlying sequence that satisfy a predicate.

The name views::filter denotes a range adaptor object (24.7.1). Given subexpressions E and P, the
expression views::filter(E, P) is expression-equivalent to filter_view{E, P}.

Example:

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

24.7.4.2 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
        // 24.7.4.3, class filter_view::iterator
        class iterator;
        // 24.7.4.4, class filter_view::sentinel
        class sentinel;
    }

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 const Pred& pred() const;

    constexpr iterator begin();
    constexpr auto end() {
        if constexpr (common_range<V>)
            return iterator{this, ranges::end(base_)};
        else
            return sentinel{this};
    }

§ 24.7.4.2 989
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 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.

24.7.4.3 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(ranges::iter_move(i.current_));
    friend constexpr void iter_swap(const iterator& x, const iterator& y)
      noexcept(ranges::iter_swap(x.current_, y.current_))
      requires indirectly_swappable<iterator_t<V>>;
  };
  }
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.

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.

(2.3) Otherwise, iterator_concept denotes input_iterator_tag.

iterator::iterator_category is defined as follows:

(3.1) Let \( C \) denote the type iterator_traits<iterator_t<V>>::iterator_category.

(3.2) If \( C \) models derived_from<bidirectional_iterator_tag>, then iterator_category denotes bidirectional_iterator_tag.

(3.3) Otherwise, if \( C \) models derived_from<forward_iterator_tag>, then iterator_category denotes forward_iterator_tag.

(3.4) Otherwise, iterator_category denotes \( C \).

constexpr iterator(filter_view& parent, iterator_t<V> current);

Effects: Initializes current_ with std::move(current) and parent_ with addressof(parent).

constexpr iterator_t<V> base() const &
requires copyable<iterator_t<V>>;

Effects: Equivalent to: return current_;

constexpr iterator_t<V> base() &&;

Effects: Equivalent to: return std::move(current_);

constexpr range_reference_t<V> operator*() const;

Effects: Equivalent to: return *current_;

constexpr iterator_t<V> operator->() const
requires has-arrow<iterator_t<V>> && copyable<iterator_t<V>>;

Effects: Equivalent to: return current_;

constexpr iterator& operator++();

Effects: Equivalent to:

\[
\text{current}_\_ = \text{ranges::find_if}(\text{std::move}(++\text{current}_\_), \text{ranges::end}(\text{parent}\_\_->\text{base}_\_), \text{ref}(*\text{parent}\_\_->\text{pred}_\_));
\]

return *this;

constexpr void operator++(int);

Effects: Equivalent to +++this.

constexpr iterator operator++(int) requires forward_range<V>;

Effects: Equivalent to:

auto tmp = *this;
++this;
return tmp;

constexpr iterator& operator--() requires bidirectional_range<V>;

Effects: Equivalent to:

\[
\text{do}
\]
\begin{align*}
\text{while} & (\text{!invoke}(\text{parent}\_\_->\text{pred}_\_, *\text{current}_\_));
\text{return} & *\text{this};
\end{align*}

constexpr iterator operator--(int) requires bidirectional_range<V>;

Effects: Equivalent to:
auto tmp = *this;
--*this;
return tmp;

friend constexpr bool operator==(const iterator& x, const iterator& y)
requires equality_comparable<iterator_t<V>>;
Effects: Equivalent to: return x.current_ == y.current_;

friend constexpr range_rvalue_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_).

24.7.4.4 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);
            constexpr 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_;

24.7.5 Transform view

24.7.5.1 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 (24.7.1). Given subexpressions E and F, the expression views::transform(E, F) is expression-equivalent to transform_view{E, F}.

[Example:

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]
24.7.5.2 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:
            // 24.7.5.3, class template transform_view::iterator
            template<bool> struct iterator; // exposition only
            // 24.7.5.4, 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 iterator<false> end();
            constexpr iterator<false> end() requires common_range<V> &
                regular_invocable<const F&, range_reference_t<const V>>;

            constexpr sentinel<false> end();
            constexpr sentinel<true> end() const
                requires common_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_); }
    };

    template<class R, class F>
    transform_view(R&&, F) -> transform_view<views::all_t<R>, F>;
}

constexpr transform_view(V base, F fun);

1 Effects: Initializes base_ with std::move(base) and fun_ with std::move(fun).

constexpr iterator<false> begin();

2 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>>;

3 Effects: Equivalent to:
    return iterator<true>{*this, ranges::begin(base_)};

constexpr sentinel<false> end();

4 Effects: Equivalent to:

§ 24.7.5.2 993
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_)};

24.7.5.3 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(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 invoke(*parent_->fun_, *current_); }
            constexpr iterator& operator++();
            constexpr void operator++(int);
            constexpr iterator operator++(int) requires forward_range<Base>;

            constexpr iterator& operator--();
            constexpr void operator--(int);
            constexpr iterator operator--(int) requires bidirectional_range<Base>;

            constexpr iterator& operator+=(difference_type n)
                requires random_access_range<Base>;

§ 24.7.5.3
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<V>, 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 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(sen
    t_t<
    Base> end);

Effects: Initializes end_ with end.

constexpr sentinel(sen
    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_
;

friend constexpr bool operator==(const 
    iterator<
    Const>& x, const 
    sentinel& y);

Effects: Equivalent to: return x.
    current_ == y.end_;

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.end_;

friend constexpr range_difference_t<
    Base>
operator-(const sentinel& y, const iterator<
    Const>& x)
    requires sized_sentinel_for<
    sentinel_t<
    Base>>, iterator_t<
    Base>>;

Effects: Equivalent to: return y.
    end_ - x.current_;

24.7.6 Take view

24.7.6.1 Overview

take_view produces a view of the first N elements from another view, or all the elements if the adapted view contains fewer than N.

The name views::take denotes a range adaptor object (24.7.1). Let E and F be expressions, let T be remove_cvref_t<decltype((E))>, and let b be range_difference_t<decltype((E))>. If decltype((F)) does not model convertible_to<D>, views::take(E, F) is ill-formed. Otherwise, the expression views::take(E, F) is expression-equivalent to:

(2.1) If T is a specialization of ranges::empty_view (24.6.1.2), then ((void) F, decay-copy(E)).

(2.2) Otherwise, if T models random_access_range and sized_range and is

(2.2.1) a specialization of span (22.7.3) where T::extent == dynamic_extent,

(2.2.2) a specialization of basic_string_view (21.4),

(2.2.3) a specialization of ranges::iota_view (24.6.3.2), or

(2.2.4) a specialization of ranges::subrange (24.5.3),

then T{ranges::begin(E), ranges::begin(E) + min<D>(ranges::size(E), F)}, except that E is evaluated only once.

(2.3) Otherwise, ranges::take_view{E, F}.

[Example:

vector<int> is{0,1,2,3,4,5,6,7,8,9};
take_view few{is, 5};
for (int i : few)
    cout << i << ' '; // prints: 0 1 2 3 4

— end example]
24.7.6.2 Class template take_view

namespace std::ranges {
    template<view V> class take_view : public view_interface<take_view<V>> {
        private:
            V base_ = V();  // exposition only
            range_difference_t<V> count_ = 0;  // exposition only
        // 24.7.6.3, class template take_view::sentinel
        template<bool> struct sentinel;  // exposition only
        public:
            take_view() = default;
            constexpr take_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 (!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);

1 Effects: Initializes base_ with std::move(base) and count_ with count.

24.7.6.3 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_t<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_t<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_;

24.7.7 Take while view

24.7.7.1 Overview

1 Given a unary predicate pred and a view r, take_while_view produces a view of the range [begin(r), ranges::find_if_not(r, pred)).

2 The name views::take_while denotes a range adaptor object (24.7.1). Given subexpressions E and F, the expression views::take_while(E, F) is expression-equivalent to take_while_view{E, F}.

3 Example:
    auto input = istringstream("0 1 2 3 4 5 6 7 8 9");
    auto small = [] (const auto x) noexcept { return x < 5; };
    auto small_ints = istream_view<int>(input) | views::take_while(small);
for (const auto i : small_ints) {
    cout << i << ' '; // prints 0 1 2 3 4
}
auto i = 0;
input >> i;
cout << i; // prints 6

— end example]

24.7.7.2 Class template `take_while_view` [range.take.while.view]

```cpp
namespace std::ranges {
    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 : public view_interface<take_while_view<V, Pred>> {
        // 24.7.7.3, class template `take_while_view::sentinel`
    }
}
```

```cpp
constexpr take_while_view(V base, Pred pred);
```

1. **Effects:** Initializes `base_` with `std::move(base)` and `pred_` with `std::move(pred)`.

```cpp
constexpr const Pred& pred() const;
```

2. **Effects:** Equivalent to: `return *pred_`.

24.7.7.3 Class template `take_while_view::sentinel` [range.take.while.sentinel]

```cpp
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 {
        // exposition only
    }
}
```
```cpp
sentinel_t<Base> end = sentinel_t<Base>(); // exposition only
const Pred* pred = nullptr; // exposition only
public:
    sentinel() = default;
constexpr explicit sentinel(sentinel_t<Base> end, const Pred* pred);
constexpr sentinel(sentinel_t<!Const> s)
    requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
constexpr sentinel_t<Base> base() const { return end_; }
fRIEND constexpr bool operator==(const iterator_t<Base>& x, const sentinel& y);

#endif
```
constexpr V base() const requires copy_constructible<V> { return base_; }
constexpr V base() && { return std::move(base_); }

constexpr auto begin() requires (!simple_view<V> && random_access_range<V>));
constexpr auto begin() const
    requires random_access_range<const V>;

constexpr auto end() requires (!
    simple_view<V>)
    { return ranges::end(base_); }
constexpr auto end() const
    requires range<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: Without this, applying a reverse_view over a drop_view would have
    quadratic iteration complexity. — end note]

24.7.9 Drop while view [range.drop.while]
24.7.9.1 Overview [range.drop.while.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 (24.7.1). Given subexpressions E and F, the
    expression views::drop_while(E, F) is expression-equivalent to drop_while_view(E, F).
3  [Example:
    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
    }
  ]

24.7.9.2 Class template drop_while_view

namespace std::ranges {

    template<view V, class Pred>
    requires input_range<V> && is_object_v<Pred> &&
    indirect_unary_predicate<const Pred, iterator_t<V>>
    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 drop_while_view(V base, Pred pred);

1  Effects: Initializes base_ with std::move(base) and pred_ with std::move(pred).

constexpr const Pred& pred() const;

2  Effects: Equivalent to: return *pred_;

constexpr auto begin();

3  Returns: ranges::find_if_not(base_, cref(*pred_)).

4  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: Without this, applying a reverse_view over a drop_while_view would have quadratic iteration complexity. — end note]

24.7.10 Join view

24.7.10.1 Overview

join_view flattens a view of ranges into a view.

2  The name views::join denotes a range adaptor object (24.7.1). Given a subexpression E, the expression views::join(E) is expression-equivalent to join_view(E).

3  [Example:
    vector<string> ss("hello", " ", "world", "!");
    join_view greeting(ss);
for (char ch : greeting)
    cout << ch; // prints: hello world!

— end example]}

24.7.10.2 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>;
      // 24.7.10.3, class template join_view::iterator
      template<bool Const>
      struct iterator;
      // exposition only
      // 24.7.10.4, 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) {
      V base() const& requires copy_constructible<V> { return base_; }
      constexpr V base() && { return std::move(base_); }
      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<simple_view<V>>{*this, ranges::end(base_)};
        else
          return sentinel<simple_view<V>>{*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);

Effects: Initializes base_ with std::move(base).

24.7.10.3 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 = // exposition only
                is_reference_v<range_reference_t<Base>>;

            iterator_t<Base> outer_ = iterator_t<Base>();// exposition only
            iterator_t<range_reference_t<Base>> inner_ = // exposition only
                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--();
            requires 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> && 
  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& i) 
  noexcept(noexcept(ranges::iter_move(i."
  return ranges::iter_move(i."
}) 

friend constexpr void iter_swap(const iterator& x, const iterator& y) 
  noexcept(noexcept(ranges::iter_swap(x."
});

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:

  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

Effects: Equivalent to:

  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;
}
if constexpr (ref-is-glvalue)
    inner_ = iterator_t<range_reference_t<Base>>();

constexpr iterator (Parent& parent, iterator_t<Base> outer);

Effects: Initializes outer_ with std::move(outer) and parent_ with addressof(parent); then calls satisfy().

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>>>;

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_.

constexpr iterator& operator++();

Effects: Let inner-range be:

- If ref-is-glvalue is true, *outer_.
- Otherwise, parent_->inner_.

Effects: Equivalent to:

auto& inner_rng = inner-range;
if (++inner_ == ranges::end(inner_rng)) {
    ++outer_;
    satisfy();
}
return *this;

constexpr void operator++(int);

Effects: Equivalent to: +++this.

constexpr iterator operator++(int)
    requires ref-is-glvalue && forward_range<Base> &&
    forward_range<range_reference_t<Base>>;

Effects: Equivalent to:

auto tmp = *this;
+++this;
return tmp;

constexpr iterator& operator--();

Effects: Equivalent to:

if (outer_ == ranges::end(parent_->base_))
    inner_ = ranges::end(*--outer_);
while (inner_ == ranges::begin(*outer_))
    inner_ = ranges::end(*--outer_);
--inner_;
return *this;

constexpr iterator operator--(int)
    requires ref-is-glvalue && bidirectional_range<Base> &&
    bidirectional_range<range_reference_t<Base>> &&
    common_range<range_reference_t<Base>>;

Effects: Equivalent to:

§ 24.7.10.3
auto tmp = *this;
--*this;
return tmp;

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>>>;

Effects: Equivalent to: return x.outer_ == y.outer_ && x.inner_ == y.inner_;

friend constexpr void iter_swap(const iterator& x, const iterator& y)
optimized(noexcept(ranges::iter_swap(x.inner_, y.inner_)));

Effects: Equivalent to: return ranges::iter_swap(x.inner_, y.inner_);

24.7.10.4 Class template join_view::sentinel

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>::sentinel {
        private:
            using Parent = // exposition only
                conditional_t<Const, const join_view, join_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(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);
    }
}

constexpr explicit sentinel(Parent& parent);

Effects: Initializes end_ with ranges::end(parent.base_).

constexpr sentinel(sentinel<!Const> s)
requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;

Effects: Initializes end_ with std::move(s.end_).

friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);

Effects: Equivalent to: return x.outer_ == y.end_;
24.7.11.2 Class template split_view

namespace std::ranges {
    template<auto> struct require-constant; // exposition only

template<class R>
    concept tiny-range = // exposition only
        sized_range<R> &&
        requires { typename require-constant<remove_reference_t<R>::size()>::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>> { // exposition only
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>

    template<bool> struct outer-iterator; // exposition only

    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>>> // exposition only
        constexpr split_view(R&& r, range_value_t<R> e);

    // exposition only
    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> { // exposition only
        return outer-iterator<true>*{*this, ranges::begin(base_)};
    }

    constexpr auto end() requires forward_range<V> && common_range<V> { // exposition only
        return outer-iterator<true>*{*this, ranges::end(base_)};
    }

    constexpr auto end() const { // exposition only
        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;
    }

§ 24.7.11.2 1010
template<class R, class P>
    split_view(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);

1 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_type_t<R>>>[range.split.outer]
    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)}.

24.7.11.3 Class template split_view::outer-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>)
        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>();

                public:
                    using iterator_concept = // 24.7.11.4, class split_view::outer-iterator::value_type
                        conditional_t<forward_range<Base>, forward_iterator_tag, input_iterator_tag>;
                    using iterator_category = input_iterator_tag;
                    struct 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 value_type operator*() const;
                    constexpr outer-iterator& operator++();
                    constexpr decltype(auto) operator++(int) {
                        if constexpr (forward_range<Base>) {
                            auto tmp = *this;
                            ++this;
                            return tmp;
                        } else
                            ++this;
                    }
        };
};
Many of the following specifications refer to the notional member \texttt{current} of \texttt{outer-iterator}. \texttt{current} is equivalent to \texttt{current\_if} \texttt{V} models \texttt{forward\_range}, and \texttt{parent\_\rightarrow current\_} otherwise.

\begin{verbatim}
constexpr explicit outer-iterator(Parent\& parent)
    requires (!forward_range<Base>);

    Effects: Initializes \texttt{parent\_} with \texttt{addressof(parent)}.
\end{verbatim}

\begin{verbatim}
constexpr outer-iterator(Parent\& parent, iterator_t<Base> current)
    requires forward_range<Base>;

    Effects: Initializes \texttt{parent\_} with \texttt{addressof(parent)} and \texttt{current\_} with \texttt{std::move(current)}.
\end{verbatim}

\begin{verbatim}
constexpr outer-iterator(const outer-iterator<!Const> i)
    requires Const \&\& convertible_to<iterator_t<V>, iterator_t<Base>>;

    Effects: Initializes \texttt{parent\_} with \texttt{i.parent\_} and \texttt{current\_} with \texttt{std::move(i.current\_)}. 
\end{verbatim}

\begin{verbatim}
constexpr value_type operator*() const;

    Effects: Equivalent to: \texttt{return value_type{*this}};
\end{verbatim}

\begin{verbatim}
constexpr outer-iterator& operator++();

    Effects: Equivalent to:
    const auto end = ranges::end(parent\_\rightarrow base\_);
    if (current == end) return *this;
    const auto [pbegin, pend] = subrange{parent\_\rightarrow 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;
\end{verbatim}

\begin{verbatim}
friend constexpr bool operator==(const outer-iterator& x, const outer-iterator& y)
    requires forward_range<Base>;

    Effects: Equivalent to: \texttt{return x.current\_ == y.current\_}.
\end{verbatim}

\begin{verbatim}
friend constexpr bool operator==(const outer-iterator& x, default_sentinel_t);

    Effects: Equivalent to: \texttt{return x.current\_ == ranges::end(x.parent\_\rightarrow base\_)};
\end{verbatim}
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>);
3 Effects: Equivalent to: return inner_iterator<Const>{std::move(i_)};

constexpr default_sentinel_t end() const;
4 Effects: Equivalent to: return default_sentinel;

24.7.11.5 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>)
    template<bool Const>
    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);
            }

§ 24.7.11.5 1013
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>>;
};
}

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.

constexpr explicit inner_iterator(outer_iterator<Const> i);

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_;
      do {
          if (*cur != *pcur) return false;
          if (++pcur == pend) return true;
          } while (++cur != end);
      return false;
  } 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).

24.7.12 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 (16.4.2.2.6). 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: 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:

1. If T models `contiguous_iterator`, then `span{to_address(E), static_cast<D>(F)}`.
2. Otherwise, if T models `random_access_iterator`, then `subrange(E, E + static_cast<D>(F))`, except that E is evaluated only once.
3. Otherwise, `subrange(counted_iterator{E, F}, default_sentinel)`. 

### 24.7.13 Common view

#### 24.7.13.1 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.

2. [Note: `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 (24.7.1). 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}`.

4. [Example:

```cpp
// 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---

#### 24.7.13.2 Class template common_view

```
namespace std::ranges {

    template<typename 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:
        common_view() = default;
        constexpr explicit common_view(V r);
    } 

    template<typename 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_);
  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);
1  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);
2  Effects: Initializes base_ with views::all(std::forward<R>(r)).

24.7.14 Reverse view
[range.reverse]
24.7.14.1 Overview
[range.reverse.overview]
1 reverse_view takes a bidirectional view and produces another view that iterates the same elements in reverse order.

2 The name views::reverse denotes a range adaptor object (24.7.1). 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-qualified

  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).

3 [Example:

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]

24.7.14.2 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();

§ 24.7.14.2
constexpr auto end() const requires common_range<const V>;

Effects: Equivalent to: return make_reverse_iterator(ranges::begin(base_));

24.7.15 Elements view

24.7.15.1 Overview

elements_view takes a view of tuple-like values and a size_t, and produces a view with a value-type of the \( N \)\(^{th} \) element of the adapted view’s value-type.

The name \( \text{views}::\text{elements}<N> \) denotes a range adaptor object (24.7.1). Given a subexpression \( E \) and constant expression \( N \), the expression \( \text{views}::\text{elements}<N>(E) \) is expression-equivalent to \( \text{elements_view<views::all_t<\text{decltype((E))}>, N>\{E\}} \).

[Example:

```cpp
auto historical_figures = map{
    {"Lovelace"sv, 1815},
    {"Turing"sv, 1912},
    {"Babbage"sv, 1791},
    {"Hamilton"sv, 1936}
};

auto names = historical_figures | views::elements<0>;
for (auto& name : names) {
    cout << name << ' ' << '1'; // prints Babbage Hamilton Lovelace Turing
}

auto birth_years = historical_figures | views::elements<1>;
for (auto& born : birth_years) {
    cout << born << ' ' << '2'; // prints 1791 1936 1815 1912
}
```

—end example]

keys_view is an alias for \( \text{elements_view<views::all_t<R>, 0>} \), and is useful for extracting keys from associative containers.

[Example:

```cpp
auto names = keys_view<historical_figures>;
for (auto& name : names) {
    cout << name << ' ' << '3'; // prints Babbage Hamilton Lovelace Turing
}
```

—end example]

values_view is an alias for \( \text{elements_view<views::all_t<R>, 1>} \), and is useful for extracting values from associative containers.

[Example:

```cpp
auto is_even = [] (const auto x) { return x % 2 == 0; };
cout << ranges::count_if(values_view<historical_figures>, is_even); // prints 2
```

—end example]

24.7.15.2 Class template elements_view

```cpp
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() requires common_range<V>
        { return sentinel<false>{ranges::end(base_)}; }
      constexpr auto end() const requires range<const V>
        { return sentinel<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:
    // 24.7.15.3, class template elements_view::iterator
    template<bool> struct iterator;
    // exposition only
    // 24.7.15.4, 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).

24.7.15.3 Class template elements_view::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>
  class elements_view<V, N>::iterator {
    template<bool Const>
    using Base = conditional_t<Const, const V, V>;
    // exploitation only
    iterator_t<Base> current_ = iterator_t<Base>();
  public:
    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_); }

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& y, 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>;
};

§ 24.7.15.3

Effects: Initializes current_ with std::move(current).

Effects: Initializes current_ with std::move(i.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:
--current_; return *this;

constexpr iterator operator--(int) requires bidirectional_range<Base>;

Effects: Equivalent to:
auto temp = *this;
--current_; return temp;

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<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+(const iterator& x, difference_type y)  
   requires random_access_range<Base>;  
   Effects: Equivalent to: return iterator{x} += y;

friend constexpr iterator operator+(difference_type x, const iterator& y)  
   requires random_access_range<Base>;  
   Effects: Equivalent to: return y + x;

constexpr iterator operator-(const iterator& x, difference_type y)  
   requires random_access_range<Base>;  
   Effects: Equivalent to: return iterator{x} -= y;

constexpr difference_type operator-(const iterator& x, const iterator& y)  
   requires random_access_range<Base>;  
   Effects: Equivalent to: return x.current_ - y.current_;

§ 24.7.15.4  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 {  // exposition only
            private:
               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> 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>>;

                  friend constexpr range_difference_t<Base>  
                     operator-(const sentinel& x, const iterator<Const>& y)  
                     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> other)  
               requires Const && convertible_to.sentinel_t<V>, sentinel_t<Base>>;
               Effects: Initializes end_ with std::move(other.end_).
constexpr sentinel_t<Base> base() const;

Effects: Equivalent to: return end_

friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);

Effects: Equivalent to: return x.current_ == y.end_

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.end_

friend constexpr range_difference_t<Base> operator-(const sentinel& x, const iterator<Const>& y)

requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;

Effects: Equivalent to: return x.end_ - 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 91: Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.2</td>
<td>Algorithms requirements</td>
</tr>
<tr>
<td>25.3</td>
<td>Parallel algorithms</td>
</tr>
<tr>
<td>25.6</td>
<td>Non-modifying sequence operations</td>
</tr>
<tr>
<td>25.7</td>
<td>Mutating sequence operations</td>
</tr>
<tr>
<td>25.8</td>
<td>Sorting and related operations</td>
</tr>
<tr>
<td>25.10</td>
<td>Generalized numeric operations</td>
</tr>
<tr>
<td>25.11</td>
<td>Specialized algorithms</td>
</tr>
<tr>
<td>25.12</td>
<td>C library algorithms</td>
</tr>
</tbody>
</table>

25.2 Algorithms requirements

1 All of the algorithms are separated from the particular implementations of data structures and are parameter-
ized 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 (6.5.2). When found by unqualified (6.5.1) name lookup for the postfix-expression in a function
call (7.6.1.2), they inhibit argument-dependent name lookup.

[Example):

```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 (13.7.6.2) 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 (23.3.5.2).

(4.2) — If an algorithm’s template parameter is named OutputIterator, OutputIterator1, or Output-
Iterator2, the template argument shall meet the Cpp17OutputIterator requirements (23.3.5.3).

(4.3) — If an algorithm’s template parameter is named ForwardIterator, ForwardIterator1, or Forward-
Iterator2, the template argument shall meet the Cpp17ForwardIterator requirements (23.3.5.4).

(4.4) — If an algorithm’s template parameter is named NoThrowForwardIterator, the template argument shall
meet the Cpp17ForwardIterator requirements (23.3.5.4), and is required to have the property that no
exceptions are thrown from increment, assignment, or comparison of, or indirection through, valid iterators.

(4.5) If an algorithm’s template parameter is named `BidirectionalIterator`, `BidirectionalAlgorithm`, or `BidirectionalAlgorithm2`, the template argument shall meet the `Cpp17BidirectionalAlgorithm` requirements (23.3.5.5).

(4.6) If an algorithm’s template parameter is named `RandomAccessAlgorithm`, `RandomAccessAlgorithm1`, or `RandomAccessAlgorithm2`, the template argument shall meet the `Cpp17RandomAccessAlgorithm` requirements (23.3.5.6).

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 algorithm shall meet the requirements of a mutable iterator (23.3). [Note: 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 (20.14.5) 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` 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)`.

8 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 boolean contexts. Unless otherwise specified, `BinaryPredicate` always takes the first iterator’s `value_type` as its first argument, that is, in those cases when `T` or `value` is part of the signature, it shall 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, v)` shall be a valid expression that is equal to `binary_pred(*first1, *first2)`, and `binary_pred(u, *first2)` shall be a valid expression that is equal to `binary_pred(*first1, *first2)`. The parameters `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, and `BinaryOperation2` are used whenever an algorithm expects a function object (20.14).

9 [Note: Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object such as `reference_wrapper<T>` (20.14.5), or some equivalent solution. —end note]

10 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.

11 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

```
auto tmp = a;
for (; n < 0; ++n) --tmp;
for (; n > 0; --n) ++tmp;
return tmp;
```

230 The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

§ 25.2 1025
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

\[
\text{iter_difference_t<decltype(a)>} n = 0;
\text{for (auto tmp = a; tmp != b; ++tmp) ++n; return n;}
\]

and if \([b, a)\) denotes a range, the same as those of

\[
\text{iter_difference_t<decltype(b)>} n = 0;
\text{for (auto tmp = b; tmp != a; ++tmp) --n; return n;}
\]

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. [Note: Consequently, the algorithms may not be called with explicitly-specified template argument lists. —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 \(\text{ExecutionPolicy}\).

Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

1. All operations of the categories of the iterators that the algorithm is instantiated with.
2. Operations on those sequence elements that are required by its specification.
3. User-provided function objects to be applied during the execution of the algorithm, if required by the specification.
4. Operations on those function objects required by the specification. [Note: See 25.2. —end note]

These functions are herein called element access functions. [Example: The \text{sort} function may invoke the following element access functions:

1. Operations of the random-access iterator of the actual template argument (as per 23.3.5.6), as implied by the name of the template parameter \(\text{RandomAccessIterator}\).
2. The \text{swap} function on the elements of the sequence (as per the preconditions specified in 25.8.1.1).
3. The user-provided \text{Compare} function object.

—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. [Note: 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] [Example:

```cpp
int x = 0;
std::mutex m;
void f() {
    int a[] = {1,2};
    std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&] (int) {
        std::lock_guard<std::mutex> guard(m); // incorrect: lock_guard constructor calls m.lock()
        ++x;
    });
}
```
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. — end example

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 could be formed by the 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.

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: For example, swap, ++, --, @=, and assignments modify the object. For the assignment and @= operators, only the left argument is modified. — end note]

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: This implies that user-supplied function objects should not rely on object identity of arguments for such input sequences. Users for whom the object identity of the arguments to these function objects is important should consider using a wrapping iterator that returns a non-copied implementation object such as reference_wrapper<T> (20.14.5) or some equivalent solution. — end note]

The invocations of element access functions in parallel algorithms invoked with an execution policy object of type execution::sequenced_policy all occur in the calling thread of execution. [Note: The invocations are not interleaved; see 6.9.1. — end note]

The invocations of element access functions in parallel algorithms invoked with an execution policy object of type 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: This means that multiple function object invocations may 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::unsequenced_policy algorithm. [Note: Because 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]

The invocations of element access functions in parallel algorithms invoked with an execution policy object of type 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 (32.4.2) or jthread (32.4.3) provide concurrent forward progress guarantees (6.9.2.2), 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: It is the caller’s responsibility to ensure that the invocation does not introduce data races or deadlocks. — end note] [Example:

```c++
int a[] = {1,2};
std::atomic<int> x{0};
std::vector<int> v;
std::for_each(std::execution::par, std::begin(a), std::end(a), [x](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]
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:

```cpp
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;
    ++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: This means that multiple function object invocations may 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: 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: 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 (6.9.2.2) 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: 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.

2 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 [algorithms.parallel.overloads]

1 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: 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.
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 $O(expr)$.

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;
    }
}
```

```cpp
// 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<iterator_t<R>, Proj>> Pred>
        constexpr bool all_of(R&& r, Pred pred, Proj proj = {});
}
```

```cpp
// 25.6.2, any of
template<class InputIterator, class Predicate>
    constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    bool any_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 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
template<class InputIterator, class Predicate>
constexpr bool none_of(InputIterator first, InputIterator 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 none_of(I first, S last, Pred pred, Proj proj = {});
}

// 25.6.4, for_each
template<class InputIterator, class Function>
constexpr Function for_each(InputIterator first, InputIterator last, Function f);

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>
constexpr for_each_result<I, Fun> 
for_each(I first, S last, Fun f, Proj proj = {});
}

template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);

namespace ranges {
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<iterator_t<R>, Proj>> Fun>
constexpr for_each_n_result<borrowed_iterator_t<R>, Fun> 
for_each_n(R&& r, 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);

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, // see 25.3.5
        ForwardIterator first, ForwardIterator last,
        Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    ForwardIterator find_if_not(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 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 BinaryPredicate>
        ForwardIterator1 find_end(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator1 first1, ForwardIterator1 last1,
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<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>
  find_end(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);

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 = {});
}

// 25.6.8, 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 = {});
}

// 25.6.10, mismatch
    template<class InputIterator1, class InputIterator2>
    constexpr pair<InputIterator1, InputIterator2>
    mismatch(InputIterator1 first1, InputIterator1 last1,
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

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,
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
bool equal(ExecutionPolicy&& exec,
           ForwardIterator1 first1, ForwardIterator1 last1,
           ForwardIterator2 first2, ForwardIterator2 last2,
           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

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 = {});
}

§ 25.4
template<forward_range R1, forward_range R2, 
class Proj1 = identity, class Proj2 = identity, 
indirect_equivalence_relation<projected<iterator_t<R1>>, Proj1>, 
projected<iterator_t<R2>>, Proj2>>
Pred = ranges::equal_to>
constexpr bool is_permutation(R1&& r1, R2&& r2, Pred pred = {}, 
Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.6.13, search
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, 
// see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
class BinaryPredicate>
ForwardIterator1
search(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>
search(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>
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);
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);

§ 25.4 1037
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);
}

template<class InputIterator, class OutputIterator, class Predicate>
constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred);

namespace ranges {
  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 copy_if_result<I, O>
  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<iterator_t<R>, Proj>> Pred>
  requires indirectly_copyable<iterator_t<R>, O>
  constexpr copy_if_result<borrowed_iterator_t<R>, O>
  copy_if(R&& r, O result, Pred pred, Proj proj = {});
}

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);
  template<bidirectional_range R, bidirectional_iterator I>
  requires indirectly_copyable<iterator_t<R>, I>
  constexpr copy_backward_result<borrowed_iterator_t<R>, I>
  copy_backward(R&& r, I result);
}

// 25.7.2, move
template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last,
        OutputIterator result);

namespace ranges {
  template<class I, class O>
  using move_result = in_out_result<I, O>;
  template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Predicate>
  ForwardIterator2 move(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator1 first, ForwardIterator1 last,
            ForwardIterator2 result, Predicate pred);
}
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,
                               // see 25.3.5
                               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,
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class UnaryOperation>
ForwardIterator2
transform(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 result, UnaryOperation op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryOperation>
ForwardIterator
transform(ExecutionPolicy&& exec, // see 25.3.5
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 O,
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 proj = {});

template<input_range R, weakly_incrementable O, copy_constructible F,
class Proj = identity>
requires indirectly_writable<O, indirect_result_t<F&, 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 O, 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 O,
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, // see 25.3.5
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,
    // see 25.3.5
    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(I first, S last, 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 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>&
        constexpr I replace_if(R&& r, Pred pred, const T& new_value, Proj proj = {});
}

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,
    // see 25.3.5
    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,
    // see 25.3.5
    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&) 0, class Proj = identity>
    requires indirectly_copyable<I, O>&&
        indirect_binary_predicate<ranges::equal_to, projected<I, Proj>>, const T1*> constexpr copy
        replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
            Proj proj = {});
    template<input_range R, class T1, class T2, output_iterator(const T2&) 0, class Proj = identity>
    requires indirectly_copyable<iterator_t<R>, O>&&
        indirect_binary_predicate<ranges::equal_to,
            projected<iterator_t<R>, Proj>>, const T1*> constexpr copy
        replace_copy(R&& r, O result, const T1& old_value, const T2& new_value,
            Proj proj = {});

§ 25.4 1041
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, class O>
constexpr replace_copy_if_result<I, O>
replace_copy_if(I first, I last, O result, Pred pred, const T& new_value,
Proj proj = {});

template<class R, class T, class O, class Proj = identity,
indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
requires indirectly_copyable<borrowed_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, ForwardIterator first, ForwardIterator last, const T& value);
	namespace ranges {
  template<class T, class O, sentinel_for<O> S>
  constexpr O fill(O first, S last, const T& value);
  template<class T, class R>
  constexpr borrowed_iterator_t<R> fill(R&& r, const T& value);
}

// 25.7.7, 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);

namespace ranges {
  template<class T, class 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, class 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);

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 Proj = 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>
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_predicate<ranges::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_predicate<ranges::equal_to, projected<iterator_t<R>*, const T*>
    constexpr remove_copy_result<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 = {});

    template<class I, class O>
    using unique_copy_result = in_out_result<I, O>;

    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, BinaryPredicate pred);

    template<class ExecutionPolicy, class InputIterator, class OutputIterator>
    constexpr OutputIterator
    unique_copy(InputIterator first, InputIterator last, OutputIterator result,
                ExecutionPolicy&& exec);
OutputIterator result, BinaryPredicate pred);

```cpp
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
```cpp
template<class BidirectionalIterator>
constexpr void reverse(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);
}
```

```
namespace ranges {
    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<borrowed_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);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator rotate(ExecutionPolicy&& exec,
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);

namespace ranges {
    template<permutable I, sentinel_for<I> S>
        rotate(I first, I middle, S last);
    template<forward_range R>
        requires permutable<iterator_t<R>>
                borrowed_subrange_t<R> rotate(R&& r, iterator_t<R> middle);
}

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);

namespace ranges {
    template<input_iterator I, sentinel_for<I> S,
        weakly_incrementable O, class Gen>
        requires (forward_iterator<I> || random_access_iterator<O>)
                using rotate_copy_result = in_out_result<I, O>;
    }
indirectly_copyable<I, O> &&
uniform_random_bit_generator<remove_reference_t<Gen>>
0 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>>
0 sample(R&& r, O out, range_difference_t<R> n, Gen&& g);

} // 25.7.13, shuffle

namespace ranges {

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);
}

// 25.7.14, shift

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); // see 25.3.5

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); // see 25.3.5

// 25.8, sorting and related operations
// 25.8.1, sorting

template<class RandomAccessIterator>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator>
void sort(ExecutionPolicy&& exec, RandomAccessIterator first, RandomAccessIterator last); // see 25.3.5

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(ExecutionPolicy&& exec, RandomAccessIterator first, RandomAccessIterator last,
Compare comp); // see 25.3.5
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
    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>
    sort(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
  template<class RandomAccessIterator>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                     Compare comp);
  template<class ExecutionPolicy, class RandomAccessIterator>
    void stable_sort(ExecutionPolicy&& exec, // see 25.3.5
                     RandomAccessIterator first, RandomAccessIterator last);
  template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
    void stable_sort(ExecutionPolicy&& exec, // see 25.3.5
                     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
    stable_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>
    stable_sort(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
  template<class RandomAccessIterator>
    constexpr void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                                 RandomAccessIterator last);
  template<class RandomAccessIterator, class Compare>
    constexpr void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                                 RandomAccessIterator last, Compare comp);
  template<class ExecutionPolicy, class RandomAccessIterator>
    void partial_sort(ExecutionPolicy&& exec, // see 25.3.5
                      RandomAccessIterator first, RandomAccessIterator middle,
                      RandomAccessIterator last);
  template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
    void partial_sort(ExecutionPolicy&& exec, // see 25.3.5
                      RandomAccessIterator first, RandomAccessIterator middle,
                      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
    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, Comp comp = {}, Proj proj = {});
}
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);

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);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
bool is_sorted(ExecutionPolicy&& exec,
// 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 bool is_sorted(I first, S last, Comp comp = {}, Proj proj = {});

template<forward_range 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 = {});

}  // namespace ranges

// 25.8.2, Nth element
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 = {});

}  // namespace ranges

§ 25.4
// 25.8.3, binary search

```cpp
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 = {});
}
```
template<class ForwardIterator, class T>
constexpr bool
binary_search(ForwardIterator first, ForwardIterator last,
    const T& value);

namespace ranges {
    template<class ForwardIterator, class T, class Compare>
constexpr bool
binary_search(ForwardIterator first, ForwardIterator last,
    const T& value, Compare comp);
}

// 25.8.4, partitions
template<class InputIterator, class Predicate>
constexpr bool
is_partitioned(InputIterator first, InputIterator 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
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 = {});
}

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
partition(ForwardIterator first,
        ForwardIterator last,
        Predicate pred);

namespace ranges {
    template<permutable 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<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);
template<class ExecutionPolicy, class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(ExecutionPolicy&& exec,
    // see 25.3.5
    BidirectionalIterator first,
    BidirectionalIterator last,
    Predicate pred);

namespace ranges {
    template<bidirectional_iterator I, sentinel_for<I> S, class Proj = identity,
        indirect_unary_pred<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_pred<projected_iterator_t<R>, Proj>> Pred>
    requires permutable<iterator_t<R>>
    borrowed_subrange_t<R> stable_partition(R&& r, Pred pred, Proj proj = {});
}

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,
    // see 25.3.5
    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_pred<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_pred<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 = {});
}

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_pred<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_pred<projected_iterator_t<R>, Proj>> Pred>
    constexpr borrowed_iterator_t<R>
    partition_point(R&& r, Pred pred, Proj proj = {});
// 25.8.5, 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);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class ForwardIterator>
ForwardIterator
merge(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    ForwardIterator result); // see 25.3.5

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);

template<class ExecutionPolicy, class BidirectionalIterator>
void inplace_merge(ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last); // see 25.3.5

template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
void inplace_merge(ExecutionPolicy&& exec,
    BidirectionalIterator first,
    BidirectionalIterator middle,
    BidirectionalIterator last, Compare comp); // see 25.3.5
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 = {});
}

// 25.8.6, set operations
template<class InputIterator1, class InputIterator2>
constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
constexpr bool includes(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 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>
ForwardIterator
set_union(ExecutionPolicy&& exec,
          // see 25.3.5
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 first2, ForwardIterator2 last2,
          Compare comp);

§ 25.4 1055
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class ForwardIterator, class Compare>
  ForwardIterator
  set_intersection(ExecutionPolicy&& exec,  // see 25.3.5
      ForwardIterator1 first1, ForwardIterator1 last1,
      ForwardIterator2 first2, ForwardIterator2 last2,
      ForwardIterator result, Compare comp);

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>
    requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
  constexpr set_intersection_result<I1, I2, O>
    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>
    requires mergeable<iterator_t<R1>, iterator_t<R2>, O, Comp, Proj1, Proj2>
  constexpr set_intersection_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O>
    set_intersection(R1&& r1, R2&& r2, O result, Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
}

template<class InputIterator1, class InputIterator2, class OutputIterator>
  constexpr OutputIterator
  set_intersection(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result);

template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
  constexpr OutputIterator
  set_intersection(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result, Compare comp);

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>
    requires mergeable<I1, I2, O, Comp, Proj1, Proj2>
  constexpr set_intersection_result<I1, I2, O>
    set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, O result,
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>, 0, Comp, Proj1, Proj2>
constexpr set_intersection_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O>
set_intersection(R1&& r1, R2&& r2, O result,
        Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
}
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class ForwardIterator>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class ForwardIterator, class Compare>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec, // see 25.3.5
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, O result, Comp comp = {},
                           Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.8.7, heap operations
template<class RandomAccessIterator>
constexpr void push_heap(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
  push_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>
  push_heap(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
  template<class RandomAccessIterator, class Compare>
  constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                          Compare comp);
  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);

§ 25.4 1058
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>
        requires sortable<iterator_t<R>, Comp, Proj>
        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>
        requires sortable<iterator_t<R>, Comp, Proj>
        constexpr borrowed_iterator_t<R> make_heap(R&& r, Comp comp = {}, Proj proj = {});
}

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<random_access_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>
        requires sortable<iterator_t<R>, Comp, Proj>
        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,
            RandomAccessIterator first, RandomAccessIterator last); // see 25.3.5
    template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
        bool is_heap(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last, // see 25.3.5
            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 = {});
}

namespace ranges {
  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);
}

namespace ranges {
  template<class RandomAccessIterator>  
  constexpr RandomAccessIterator
  is_heap_until(ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last);
  template<class ExecutionPolicy, class RandomAccessIterator, class Compare>  
  constexpr RandomAccessIterator
  is_heap_until(ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
}

namespace ranges {
  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<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr borrowed_iterator_t<R>
  is_heap_until(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.8.8, minimum and maximum
namespace ranges {
  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<iterator_t<R>, Proj>> Comp = ranges::less>
  constexpr range_value_t<R>
  min(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
  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 T max(const T& a, const T& b, Compare 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> t, Compare 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, Compare comp = {}, Proj proj = {});
}

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);

namespace ranges {
    template<class T, class Proj = identity,
        indirect_strict_weak_order<projected<const T*, Proj>> Comp = ranges::less>
    constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare 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, Compare comp = {}, Proj proj = {});
}

namespace ranges {
    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, Compare 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, Compare 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, Compare comp = {}, Proj proj = {});
}

template<class ForwardIterator>
constexpr ForwardIterator min_element(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, Compare 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, Compare comp = {}, Proj proj = {});
}
min_element(R&& r, Comp comp = {}, Proj proj = {});
}

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);
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator max_element(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator max_element(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last, Compare comp);

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<iterator_t<R>, Proj>> Comp = ranges::less>
        constexpr borrowed_iterator_t<R>
max_element(R&& r, Comp comp = {}, Proj proj = {});  
}

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, // see 25.3.5
ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last, Compare comp);

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 minmax_element_result<I>
        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 minmax_element_result<borrowed_iterator_t<R>>
        minmax_element(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.8.9, bounded value
template<class T>
constexpr T& clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
constexpr 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 = {});
}

// 25.8.10, 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);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
bool
lexicographical_compare(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class Compare>
bool
lexicographical_compare(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 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 = {});
}

// 25.8.11, 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));

// 25.8.12, 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 = {});
}

template<class BidirectionalIterator>
constexpr bool prev_permutation(BidirectionalIterator first,
                               BidirectionalIterator last);

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<const 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 III1, class III2>
    requires convertible_to<const I1&, III1> && convertible_to<const I2&, III2>
constexpr operator in_in_result<III1, III2>() const & {
    return {in1, in2};
}

template<class III1, class III2>
    requires convertible_to<III1, III1> && convertible_to<III2, III2>
constexpr operator in_in_result<III1, III2>() && {
    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 II1, class II2>
    requires convertible_to<const I&, II1> && convertible_to<const O&, II2>
constexpr operator in_out_result<II1, II2>() const & {
    return {in, out};
}

template<class II1, class II2>
    requires convertible_to<II1, I> && convertible_to<II2, O>
constexpr operator in_out_result<II1, II2>() && {
    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 III1, class III2, class OO>
    requires convertible_to<const I1&, III1> &&
               convertible_to<const I2&, III2> &&
               convertible_to<const O&, OO>
constexpr operator in_in_out_result<III1, III2, OO>() const & {
    return {in1, in2, out};
}

template<class III1, class III2, class OO>
    requires convertible_to<III1, I1> &&
               convertible_to<III2, I2> &&
               convertible_to<OO, O>
constexpr operator in_in_out_result<III1, III2, 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<const II&, II> &&
    convertible_to<const OO1&, OO1> &&
    convertible_to<const OO2&, 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<const I&, I2>
constexpr operator in_found_result<I2>() && {
    return {std::move(in), found};
}

25.6 Non-modifying sequence operations [alg.nonmodifying]

25.6.1 All of [alg.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, ForwardIterator first, ForwardIterator last,
Predicate pred);

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 = {});

template<input_range R, class Proj = identity,
indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::all_of(R&& r, Pred pred, Proj proj = {});

Let E be:
1. pred(*i) for the overloads in namespace std;
2. invoke(pred, invoke(proj, *i)) for the overloads in namespace ranges.

Returns: false if E is false for some iterator i in the range [first, last), and true otherwise.

Complexity: At most last - first applications of the predicate and any projection.

25.6.2 Any of

template<class InputIterator, class Predicate>
constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
Predicate pred);

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 = {});

template<input_range R, class Proj = identity,
indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::any_of(R&& r, Pred pred, Proj proj = {});

Let E be:
1. pred(*i) for the overloads in namespace std;
2. invoke(pred, invoke(proj, *i)) for the overloads in namespace ranges.

Returns: true if E is true for some iterator i in the range [first, last), and false otherwise.

Complexity: At most last - first applications of the predicate and any projection.

25.6.3 None of

template<class InputIterator, class Predicate>
constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool none_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
Predicate pred);

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 = {});

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. pred(*i) for the overloads in namespace std;
2. invoke(pred, invoke(proj, *i)) for the overloads in namespace ranges.

Returns: false if E is true for some iterator i in the range [first, last), and true otherwise.

Complexity: At most last - first applications of the predicate and any projection.
25.6.4 For each

```cpp
template<class InputIterator, class Function>
constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

1. **Preconditions:** Function meets the Cpp17MoveConstructible requirements (Table 28). [Note: Function need not meet the requirements of Cpp17CopyConstructible (Table 29). — end note]

2. **Effects:** Applies f to the result of dereferencing every iterator in the range [first, last), starting from first and proceeding to last - 1. [Note: If the type of first meets the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator. — end note]

3. **Returns:** f.

4. **Complexity:** Applies f exactly last - first times.

5. **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);
```

6. **Preconditions:** Function meets the Cpp17CopyConstructible requirements.

7. **Effects:** Applies f to the result of dereferencing every iterator in the range [first, last). [Note: If the type of first meets the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator. — end note]

8. **Complexity:** Applies f exactly last - first times.

9. **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.

10. **Note:** Does not return a copy of its Function parameter, since parallelization may 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 = {});
```

11. **Effects:** Calls invoke(f, invoke(proj, *i)) for every iterator i in the range [first, last), starting from first and proceeding to last - 1. [Note: If the result of invoke(proj, *i) is a mutable reference, f may apply non-constant functions. — end note]

12. **Returns:** {last, std::move(f)}.

13. **Complexity:** Applies f and proj exactly last - first times.

14. **Remarks:** If f returns a result, the result is ignored.

15. **Note:** The overloads in namespace ranges require Fun to model copy_constructible. — end note]

```cpp
template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
```

16. **Mandates:** The type Size is convertible to an integral type (7.3.8, 11.4.7).

17. **Preconditions:** Function meets the Cpp17MoveConstructible requirements. [Note: Function need not meet the requirements of Cpp17CopyConstructible. — end note]

18. **Preconditions:** n >= 0 is true.

19. **Effects:** Applies f to the result of dereferencing every iterator in the range [first, first + n) in order. [Note: If the type of first meets the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator. — end note]

20. **Returns:** first + n.

21. **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 (7.3.8, 11.4.7).
Preconditions: Function meets the Cpp17CopyConstructible requirements.
Preconditions: n >= 0 is true.
Effects: Applies f to the result of dereferencing every iterator in the range [first, first + n). [Note: If the type of first meets the requirements of a mutable iterator, f may 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>
constexpr 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 [first, first + n)
in order. [Note: If the result of invoke(proj, *i) is a mutable reference, f may apply non-constant
functions. — end note]
Returns: {first + n, std::move(f)}.
Remarks: If f returns a result, the result is ignored.
[Note: 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_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*> 
constexpr borrowed_iterator_t<R> ranges::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 ranges::find_if(I first, S last, Pred pred, 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. $\text{bool} (\text{invoke} (\text{proj}, *i) == \text{value})$ for ranges::find;
5. $\text{bool} (\text{invoke} (\text{pred}, \text{invoke} (\text{proj}, *i)))$ for ranges::find_if;
6. $\text{bool} (!\text{invoke} (\text{pred}, \text{invoke} (\text{proj}, *i)))$ for ranges::find_if_not.

2. Returns: The first iterator $i$ in the range $[\text{first}, \text{last})$ for which $E$ is true. Returns $\text{last}$ if no such iterator is found.

3. Complexity: At most $\text{last} - \text{first}$ applications of the corresponding predicate and any projection.

### 25.6.6 Find end

```
30.6.6 1070
```

```cpp
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);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_end(ExecutionPolicy&& exec,
  ForwardIterator1 first1, ForwardIterator1 last1,
  ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

```cpp
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 = {});
```
Let:

1. \( \text{proj1} = \{\}, \text{proj2} = \{\} \);

(1.1) \( \text{pred} \) be \textit{equal_to\{}\( \) for the overloads with no parameter \textit{pred};

(1.2) \( E \) be:

(1.2.1) \( \text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, *(i + n)), \text{invoke}(\text{proj2}, *(\text{first2} + n))) \) for the overloads in namespace \( \text{ranges} \);

(1.2.2) \( \text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, *(i + n)), \text{invoke}(\text{proj2}, *(\text{first2} + n))) \) for the overloads in namespace \( \text{std} \);

(1.3) \( i \) be \( \text{last1} \) if \( \text{[first2, last2)} \) is empty, or if \( (\text{last2} - \text{first2}) > (\text{last1} - \text{first1}) \) is \textit{true}, or if there is no iterator in the range \( \text{[first1, last1 - (last2 - first2)} \) such that for every non-negative integer \( n < (\text{last2} - \text{first2}) \), \( E \) is \textit{true}. Otherwise \( i \) is the last such iterator in \( \text{[first1, last1 - (last2 - first2)} \).

Returns:

(2.1) \( i \) for the overloads in namespace \( \text{std} \).

(2.2) \( \{i, i + (i == \text{last1} ? 0 : \text{last2 - first2})\} \) for the overloads in namespace \( \text{ranges} \).

Complexity: At most \( (\text{last2} - \text{first2}) * (\text{last1} - \text{first1} - (\text{last2} - \text{first2}) + 1) \) applications of the corresponding predicate and any projections.

### 25.6.7 Find first

[\textit{alg.find.first.of}]

```
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) \( *i == *j \) for the overloads with no parameter \textit{pred};
— \( \text{pred}(\ast i, \ast j) \neq \text{false} \) for the overloads with a parameter \( \text{pred} \) and no parameter \( \text{proj} \);

— \( \text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i), \text{invoke}(\text{proj}, \ast j))) \) for the overloads with parameters \( \text{pred} \) and \( \text{proj} \).

**Effects:** Finds an element that matches one of a set of values.

**Returns:** The first iterator \( i \) in the range \([\text{first1}, \text{last1})\) such that for some iterator \( j \) in the range \([\text{first2}, \text{last2})\) \( E \) holds. Returns \( \text{last1} \) if \([\text{first2}, \text{last2})\) is empty or if no such iterator is found.

**Complexity:** At most \((\text{last1} - \text{first1}) \ast (\text{last2} - \text{first2})\) applications of the corresponding predicate and any projections.

### 25.6.8 Adjacent find [alg.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<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> ranges::adjacent_find(R&& r, Pred pred = {}, Proj proj = {});
```

1 Let \( E \) be:

\[
\begin{align*}
&\text{(1.1)} & \ast i = \ast (i + 1) & \text{for the overloads with no parameter \( \text{pred} \);} \\
&\text{(1.2)} & \text{pred}(\ast i, \ast (i + 1)) \neq \text{false} & \text{for the overloads with a parameter \( \text{pred} \) and no parameter \( \text{proj} \);} \\
&\text{(1.3)} & \text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i), \text{invoke}(\text{proj}, \ast (i + 1)))) & \text{for the overloads with both parameters \( \text{pred} \) and \( \text{proj} \).}
\end{align*}
\]

2 **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.

3 **Complexity:** For the overloads with no \( \text{ExecutionPolicy} \), exactly

\[
\min((i - \text{first}) + 1, (\text{last} - \text{first}) - 1)
\]

applications of the corresponding predicate, where \( i \) is \( \text{adjacent_find} \)'s return value. For the overloads with an \( \text{ExecutionPolicy} \), \( O(\text{last} - \text{first}) \) applications of the corresponding predicate, and no more than twice as many applications of any projection.

### 25.6.9 Count [alg.count]

```
template<class InputIterator, class T>
constexpr typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);
```

§ 25.6.9
template<class ExecutionPolicy, class ForwardIterator, class T>
  typename iterator_traits<ForwardIterator>::difference_type
  count(ExecutionPolicy&& exec,
       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,
            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 iter_difference_t<I>
  ranges::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>
  ranges::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>
  ranges::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>
  ranges::count_if(R&& r, Pred pred, Proj proj = {});

Let $E$ be:

1. $i == \text{value}$ for the overloads with no parameter $\text{pred}$ or $\text{proj}$;
2. $\text{pred}(\*i) != \text{false}$ for the overloads with a parameter $\text{pred}$ but no parameter $\text{proj}$;
3. $\text{invoke}(\text{proj}, \*i) == \text{value}$ for the overloads with a parameter $\text{proj}$ but no parameter $\text{pred}$;
4. $\text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \*i)))$ 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]

template<class InputIterator1, class InputIterator2>
  constexpr pair<InputIterator1, InputIterator2>
  mismatch(InputIterator1 first1, InputIterator1 last1,
            InputIterator2 first2);

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>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

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<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 = {});

1 Let last2 be first2 + (last1 - first1) for the overloads with no parameter last2 or r2.
2 Let $E$ be:
   
   $\neg \left( \neg (\ast(first1 + n) == \ast(first2 + n)) \right)$ for the overloads with no parameter $\text{pred}$;
   
   $\neg \text{\text{pred}}(\ast(first1 + n), \ast(first2 + n)) == \text{false}$ for the overloads with a parameter $\text{pred}$ and
   no parameter $\text{proj1}$;
   
   $\neg \text{\text{ invoke}}(\text{pred}, \text{\text{ invoke}}(\text{proj1}, \ast(first1 + n)), \text{\text{ invoke}}(\text{proj2}, \ast(first2 + n)))$ for the
   overloads with both parameters $\text{pred}$ and $\text{proj1}$.
3 Let $N$ be $\min(last1 - first1, last2 - first2)$.
4 Returns: \{ first1 + n, first2 + n \}, where $n$ is the smallest integer in \[0, N\) such that $E$ holds,
or $N$ if no such integer exists.
5 Complexity: At most $N$ applications of the corresponding predicate and any projections.

25.6.11 Equal

template<class InputIterator1, class InputIterator2>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
bool equal(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);
template<class InputIterator1, class InputIterator2,
class BinaryPredicate>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
bool equal(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 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,
class BinaryPredicate>
bool equal(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

Let:

1. \( \text{last2} \) be \( \text{first2} + (\text{last1} - \text{first1}) \) for the overloads with no parameter \( \text{last2} \) or \( r2 \);
2. \( \text{pred} \) be \( \text{equal}_\to() \) for the overloads with no parameter \( \text{pred} \);
3. \( E \) be:
   4. \( \text{pred}(\ast i, \ast (\text{first2} + (i - \text{first1}))) \) for the overloads with no parameter \( \text{proj1} \);
   5. \( \text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, \ast i), \text{invoke}(\text{proj2}, \ast (\text{first2} + (i - \text{first1})))) \) for the overloads with parameter \( \text{proj1} \).

Returns: If \( \text{last1} - \text{first1} \) != \( \text{last2} - \text{first2} \), return \( \text{false} \). Otherwise return \( \text{true} \) if \( E \) holds for every iterator \( i \) in the range \([\text{first1}, \text{last1}]\) Otherwise, returns \( \text{false} \).

Complexity: If the types of \( \text{first1}, \text{last1}, \text{first2}, \) and \( \text{last2} \):

1. meet the \( \text{Cpp17RandomAccessIterator} \) requirements (23.3.5.6) for the overloads in namespace \( \text{std} \);
2. pairwise model \( \text{sized_sentinel_for} \) (23.3.4.8) for the overloads in namespace \( \text{ranges} \),
   and \( \text{last1} - \text{first1} \) != \( \text{last2} - \text{first2} \), then no applications of the corresponding predicate and each projection; otherwise,
3. For the overloads with no \( \text{ExecutionPolicy} \), at most \( \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \) applications of the corresponding predicate and any projections.
— For the overloads with an ExecutionPolicy, $O(\min(last1 - first1, last2 - first2))$ applications of the corresponding predicate.

### 25.6.12 Is permutation

For the overloads with an ExecutionPolicy, $O(\min(last1 - first1, last2 - first2))$ applications of the corresponding predicate.

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);

#### Mandates
ForwardIterator1 and ForwardIterator2 have the same value type.

#### Preconditions
The comparison function is an equivalence relation.

#### Remarks
If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

#### Returns
If last1 - first1 != last2 - first2, return false. Otherwise return true if there exists a permutation of the elements in the range [first2, first2 + (last1 - first1)], beginning with ForwardIterator2 begin, such that equal(first1, last1, begin) returns true or equal(first1, last1, begin, pred) returns true; otherwise, returns false.

#### 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) would return true if pred was not given in the argument list or equal(first1, last1, first2, last2, pred) would return true if pred was given in the argument list; otherwise, at worst $O(N^2)$, where $N$ has the value last1 - first1.

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 ranges::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<projected<iterator_t<R1>, Proj1>, projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to>
constexpr bool ranges::is_permutation(R1&& r1, R2&& r2, Pred pred = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

#### 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:

1. S1 and I1 model sized_sentinel_for<S1, I1>,
2. S2 and I2 model sized_sentinel_for<S2, I2>, and
3. 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

[alg.search]

```cpp
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);
```

1 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: 
   \( *(i + n) == *(first2 + n), \) \( \text{pred}(*(i + n), *(first2 + n)) \) \(!= false\). Returns \( \text{first1} \) if \([first2, last2)\) is empty, otherwise returns \( \text{last1} \) if no such iterator is found.

2 Complexity: At most \((last1 - first1) * (last2 - first2)\) applications of the corresponding predicate.

```cpp
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::search(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::search(R1&& r1, R2&& r2, Pred pred = {},
               Proj1 proj1 = {}, Proj2 proj2 = {});
```

3 Returns:

   (3.1) \{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 
   \( \text{bool(invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))))} \) 
   is true.

   (3.2) Returns \{last1, last1\} if no such iterator exists.

4 Complexity: At most \((last1 - first1) * (last2 - first2)\) 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);
```
template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
ForwardIterator
    search_n(ExecutionPolicy&& exec,
            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);

Mandates: The type Size is convertible to an integral type (7.3.8, 11.4.7).

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:
\(*(i + n) == value, pred(*(i + n), value) != false\). Returns \(last\) if no such
iterator is found.

Complexity: At most \(last - first\) applications of the corresponding predicate.

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 = {});

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);

for

Effects: Equivalent to: return searcher(first, last).first;

Remarks: Searcher need not meet the Cpp17CopyConstructible requirements.
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 \( \ast(\text{result} + n) = \ast(\text{first} + n) \).
4. **Returns:**
   - \( \text{result} + N \) for the overload in namespace \( \text{std} \).
   - \( \{\text{last}, \text{result} + N\} \) for the overloads in namespace \( \text{ranges} \).
5. **Complexity:** Exactly \( N \) assignments.

```
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 \( \ast(\text{result} + n) = \ast(\text{first} + n) \).
8. **Returns:** \( \text{result} + (\text{last} - \text{first}) \).
9. **Complexity:** Exactly \( \text{last} - \text{first} \) assignments.

```
template<class InputIterator, class Size, class OutputIterator>
constexpr OutputIterator copy_n(InputIterator first, Size n,
                                OutputIterator result);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class Size, class ForwardIterator2>
ForwardIterator2 copy_n(ExecutionPolicy&& exec,
                         ForwardIterator1 first, Size n,
                         ForwardIterator2 result);
```

```
template<input_iterator I, weakly_incrementable O>
requires indirectly_copyable<I, O>
constexpr ranges::copy_n_result<I, O>
ranges::copy_n(I first, iter_difference_t<I> n, O result);
```

10. Let \( N \) be \( \max(0,n) \).
11. **Mandates:** The type \( \text{Size} \) is convertible to an integral type (7.3.8, 11.4.7).
12. **Effects:** For each non-negative integer \( i < N \), performs \( \ast(\text{result} + i) = \ast(\text{first} + i) \).
13. **Returns:**
   - \( \text{result} + N \) for the overloads in namespace \( \text{std} \).
   - \( \{\text{first} + N, \text{result} + N\} \) for the overload in namespace \( \text{ranges} \).
14. **Complexity:** Exactly \( N \) assignments.

```
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);
```
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::copy_if_result<I, O>
  ranges::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 ranges::copy_if_result<borrowed_iterator_t<R>, O>
  ranges::copy_if(R&& r, O result, Pred pred, Proj proj = {});

15 Let $E$ be:

- (15.1) $\text{bool}(\text{pred}(\ast i))$ for the overloads in namespace std;
- (15.2) $\text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i)))$ for the overloads in namespace ranges,
and $N$ be the number of iterators $i$ in the range $[\text{first}, \text{last})$ for which the condition $E$ holds.

16 **Preconditions:** The ranges $[\text{first}, \text{last})$ and $[\text{result}, \text{result} + (\text{last} - \text{first}))$ do not overlap.

[Note: For the overload with an ExecutionPolicy, there may be a performance cost if iterator_traits<ForwardIterator>::value_type is not Cpp17MoveConstructible (Table 28). — end note]

**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 **Returns:**

- (18.1) $\text{result} + N$ for the overloads in namespace std.
- (18.2) $\{\text{last}, \text{result} + N\}$ for the overloads in namespace ranges.

19 **Complexity:** Exactly $\text{last} - \text{first}$ applications of the corresponding predicate and any projection.

20 **Remarks:** Stable (16.5.5.8).

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
  copy_backward(BidirectionalIterator1 first,
               BidirectionalIterator1 last,
               BidirectionalIterator2 result);

template<bidirectional_iterator I1, sentinel_for<I1> S1, bidirectional_iterator I2>
requires indirectly_copyable<I1, I2>
constexpr ranges::copy_backward_result<I1, I2>
  ranges::copy_backward(I1 first, S1 last, I2 result);

template<bidirectional_range R, bidirectional_iterator I>
requires indirectly_copyable<iterator_t<R>, I>
constexpr ranges::copy_backward_result<borrowed_iterator_t<R>, I>
  ranges::copy_backward(R&& r, I result);

21 Let $N$ be $\text{last} - \text{first}$.

22 **Preconditions:** $\text{result}$ is not in the range $[\text{first}, \text{last})$.

23 **Effects:** Copies elements in the range $[\text{first}, \text{last})$ into the range $[\text{result} - N, \text{result})$ starting from last - 1 and proceeding to first.\footnote{copy_backward should be used instead of copy when last is in the range $[\text{result} - N, \text{result})$.}

24 For each positive integer $n \leq N$, performs $\ast(\text{result} - n) = \ast(\text{last} - n)$.

25 **Returns:**

- (24.1) $\text{result} - N$ for the overload in namespace std.
- (24.2) $\{\text{last}, \text{result} - N\}$ for the overloads in namespace ranges.

**Complexity:** Exactly $N$ assignments.

### 25.7.2 Move

\[\text{alg.move}\]

template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last,
                             OutputIterator result);

\[\text{§ 25.7.2}\]
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
\[
\begin{align*}
(1.1) & \quad \text{std::move}(*(first + n)) \quad \text{for the overload in namespace std}; \\
(1.2) & \quad \text{ranges::iter_move}(first + n) \quad \text{for the overloads in namespace ranges}.
\end{align*}
\]

Let \( N \) be \( \text{last} - \text{first} \).

Preconditions: \( \text{result} \) 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 \text{first} and proceeding to \text{last}. For each non-negative integer \( n < N \), performs \( *(\text{result} + n) = E \).

Returns:
\[
\begin{align*}
(4.1) & \quad \text{result} + N \quad \text{for the overload in namespace std}; \\
(4.2) & \quad \{\text{last}, \text{result} + N\} \quad \text{for the overloads in namespace ranges}.
\end{align*}
\]

Complexity: Exactly \( N \) assignments.

Let \( N \) be \( \text{last} - \text{first} \).

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 \( *(\text{result} + n) = \text{std::move}(*(\text{first} + n)) \).

Returns: \( \text{result} + N \).

Complexity: Exactly \( N \) assignments.

Let \( E \) be
\[
\begin{align*}
(11.1) & \quad \text{std::move}(*(\text{last} - n)) \quad \text{for the overload in namespace std}; \\
(11.2) & \quad \text{ranges::iter_move}(\text{last} - n) \quad \text{for the overloads in namespace ranges}.
\end{align*}
\]

Let \( N \) be \( \text{last} - \text{first} \).

Preconditions: \( \text{result} \) 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}.

\[\text{move\_backward} \] should be used instead of \text{move} when \text{last} is in the range \([\text{result} - N, \text{result})\).
\( n \) = \( E \).

**Returns:**

- \( \text{result} - N \) for the overload in namespace `std`.
- \( \{\text{last}, \text{result} - N\} \) for the overloads in namespace `ranges`.

**Complexity:** Exactly \( N \) assignments.

### 25.7.3 Swap

```cpp
template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
swap_ranges(ExecutionPolicy&& exec,
            ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

```cpp
template<input_iterator I1, sentinel_for<I1> S1, input_iterator I2, sentinel_for<I2> S2>
requires indirectly_swappable<I1, I2>
constexpr ranges::swap_ranges_result<I1, I2>
ranges::swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);
```

```cpp
template<input_range R1, input_range R2>
requires indirectly_swappable<iterator_t<R1>, iterator_t<R2>>
constexpr ranges::swap_ranges_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>>
ranges::swap_ranges(R1&& r1, R2&& r2);
```

1. Let:

   - \( \text{last2} \) be \( \text{first2} + (\text{last1} - \text{first1}) \) for the overloads with no parameter named \( \text{last2} \);
   - \( M \) be \( \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \).

2. **Preconditions:** The two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) do not overlap. For the
   overloads in namespace `std`, \(*(\text{first1} + n)\) is swappable with \((16.5.3.2) \*(\text{first2} + n)\).

3. **Effects:** For each non-negative integer \( n < M \) performs:

   - \( \text{swap}(\*(\text{first1} + n), \*(\text{first2} + n)) \) for the overloads in namespace `std`;
   - \( \text{ranges::iter_swap}(\text{first1} + n, \text{first2} + n) \) for the overloads in namespace `ranges`.

4. **Returns:**

   - \( \text{last2} \) for the overloads in namespace `std`.
   - \( \{\text{first1} + M, \text{first2} + M\} \) for the overloads in namespace `ranges`.

5. **Complexity:** Exactly \( M \) swaps.

### 25.7.4 Transform

```cpp
template<class InputIterator, class OutputIterator, class UnaryOperation>
constexpr void iter_swap(InputIterator a, OutputIterator b);
```

1. **Preconditions:** \( a \) and \( b \) are dereferenceable. \( \*a \) is swappable with \((16.5.3.2) \*b\).

2. **Effects:** As if by \text{swap}(\*a, \*b).

### 25.7.4 Transform

```cpp
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator
transform(InputIterator first1, InputIterator last1,
         OutputIterator result, UnaryOperation op);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class UnaryOperation>
ForwardIterator2
transform(ExecutionPolicy&& exec,
         ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2,
         UnaryOperation op);
```
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 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 Proj1 = identity, class Proj2 = identity>
requires indirectly_writable<O, indirect_result_t<F&, projected<iterator_t<R1>, Proj1>, projected<iterator_t<R2>, Proj2>>
constexpr ranges::binary_transform_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O> ranges::transform(R&& r1, R&& 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
   1. op(*(first1 + (i - result))) for unary transforms defined in namespace std;
   2. binary_op(*(first1 + (i - result)), *(first2 + (i - result))) for binary transforms defined in namespace std;
   3. invoke(op, invoke(proj, *(first1 + (i - result)))) for unary transforms defined in namespace ranges;
   4. 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

1. [first1, first1 + N],
2. [first2, first2 + N], and
Effects: Assigns through every iterator i in the range \([\text{result}, \text{result} + N]\) a new corresponding value equal to \(E\).

Returns:

- \(\text{result} + N\) for the overloads defined in namespace \(\text{std}\).
- \(\{\text{first1} + N, \text{result} + N\}\) for unary transforms defined in namespace \(\text{ranges}\).
- \(\{\text{first1} + N, \text{first2} + N, \text{result} + N\}\) for binary transforms defined in namespace \(\text{ranges}\).

Complexity: Exactly \(N\) applications of \(\text{op}\) or \(\text{binary\_op}\), and any projections. This requirement also applies to the overload with an \(\text{ExecutionPolicy}\).

Remarks: \(\text{result}\) may be equal to \(\text{first1}\) or \(\text{first2}\).

### 25.7.5 Replace

```cpp
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 = {});

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&>
constexpr borrowed_iterator_t<R> ranges::replace_if(R&& r, Pred pred, const T& new_value, Proj proj = {});
```

233) The use of fully closed ranges is intentional.
Mandates: 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 new_value, when \(E\) is true.

Returns: last for the overloads in namespace ranges.

Complexity: Exactly last – first applications of the corresponding predicate and any projection.

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);

template<input_iterator I, sentinel_for<I> S, class T1, class T2, output_iterator<const T2*>& 0,
class Proj = identity>
requires indirectly_copyable<I, 0> &&
indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T1*>
constexpr ranges::replace_copy_result<I, O>
ranges::replace_copy(I first, S last, 0 result, const T1& old_value, const T2& new_value,
Proj proj = {});

template<input_range R, class T1, class T2, output_iterator<const T2*>& 0,
class Proj = identity>
requires indirectly_copyable<iterator_t<R>, 0> &&
indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T1*>
constexpr ranges::replace_copy_result<borrowed_iterator_t<R>, O>
ranges::replace_copy(R&& r, 0 result, const T1& old_value, const T2& new_value,
Proj proj = {});

Let \(E\) be

(6.1) \(\text{bool}(*(	ext{first} + (\text{i - result}))) == \text{old_value}\) for replace_copy;

(6.2) \(\text{bool(pred(*(	ext{first} + (\text{i - result}))))}\) for replace_copy_if;
(6.3) \[ \text{bool}(\text{invoke}(\text{proj}, *(\text{first} + (i - \text{result}))) == \text{old_value}) \text{ for } \text{ranges}::\text{replace} \_\text{copy}; \]

(6.4) \[ \text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *(\text{first} + (i - \text{result})))) \text{ for } \text{ranges}::\text{replace} \_\text{copy} \_\text{if}. \]

Mandates: The results of the expressions \(*\text{first}\) and \(\text{new_value}\) are writable (23.3.1) to \(\text{result}\).

Preconditions: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) do not overlap.

Effects: Assigns through every iterator \(i\) in the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) a new corresponding value

(9.1) \[ \text{new_value} \text{ if } E \text{ is true or} \]

(9.2) \[ *(\text{first} + (i - \text{result})) \text{ otherwise.} \]

Returns:

(10.1) \[ \text{result} + (\text{last} - \text{first}) \] for the overloads in namespace \(\text{std}\).

(10.2) \[ \{\text{last}, \text{result} + (\text{last} - \text{first})\} \] for the overloads in namespace \(\text{ranges}\).

Complexity: Exactly \(\text{last} - \text{first}\) applications of the corresponding predicate and any projection.

### 25.7.6 Fill

[alg.fill]

```cpp
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,
          ForwardIterator first, ForwardIterator last, const T& value);
```

```cpp
template<class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
ForwardIterator fill_n(ExecutionPolicy&& exec,
                       ForwardIterator first, Size n, const T& value);
```

```cpp
Let \(N\) be \(\max(0, n)\) for the \(\text{fill}_n\) algorithms, and \(\text{last} - \text{first}\) for the \(\text{fill}\) algorithms.

Mandates: The expression \(\text{value}\) is writable (23.3.1) to the output iterator. The type \(\text{Size}\) is convertible to an integral type (7.3.8, 11.4.7).

Effects: Assigns \(\text{value}\) through all the iterators in the range \([\text{first}, \text{first} + N)\).

Returns: \(\text{first} + N\).

Complexity: Exactly \(N\) assignments.

### 25.7.7 Generate

[alg.generate]

```cpp
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);
```

```cpp
template<class OutputIterator, class Size, class Generator>
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);
```

§ 25.7.7 1086
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);

template<input_or_output_iterator O, copy_constructible F>
  requires invocable<F&> && indirectly_writable<O, invoke_result_t<F&>>
  constexpr O ranges::generate_n(O first, iter_difference_t<O> n, F gen);

Let \( N \) be \( \max(0, n) \) for the \texttt{generate\_n} algorithms, and \( \text{last} - \text{first} \) for the \texttt{generate} algorithms.

\textbf{Mandates:} Size is convertible to an integral type (7.3.8, 11.4.7).

\textbf{Effects:} Assigns the result of successive evaluations of \texttt{gen()} through each iterator in the range \([\text{first}, \text{first} + N)\).

\textbf{Returns:} \text{first} + \( N \).

\textbf{Complexity:} Exactly \( N \) evaluations of \texttt{gen()} and assignments.

25.7.8 Remove

\texttt{template<class ForwardIterator, class T>
\text{constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,}
\text{ const T& value);}

\texttt{template<class ExecutionPolicy, class ForwardIterator, class T>
\text{ForwardIterator remove(ExecutionPolicy\&\& exec,}
\text{ ForwardIterator first, ForwardIterator last,}
\text{ const T& value);}

\texttt{template<class ForwardIterator, class Predicate>
\text{constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,}
\text{ Predicate pred);}

\texttt{template<class ExecutionPolicy, class ForwardIterator, class Predicate>
\text{ForwardIterator remove_if(ExecutionPolicy\&\& exec,}
\text{ ForwardIterator first, ForwardIterator last,}
\text{ Predicate pred);}

\texttt{template<permutable I, sentinel_for<I> S, class T, class Proj = identity>
\text{requires indirect_binary_predicate<ranges::\texttt{equal\_to}, projected<I, Proj>>, const T*>}
\text{constexpr subrange<I> ranges::\texttt{remove}(I first, S last, const T& value, Proj proj = {});}

\texttt{template<forward_range R, class T, class Proj = identity>
\text{requires permutable<iterator_t<R>>,}
\text{ indirect_binary_predicate<ranges::\texttt{equal\_to}, projected<iterator_t<R>, Proj>>, const T*>}
\text{constexpr borrowed_subrange_t<R> ranges::\texttt{remove}(R&& r, const T& value, Proj proj = {});}

\texttt{template<permutable I, sentinel_for<I> S, class Proj = identity,}
\text{ indirect_unary_predicate<projected<I, Proj>>, Pred>
\text{constexpr subrange<I> ranges::\texttt{remove\_if}(I first, S last, Pred pred, Proj proj = {});}

\texttt{template<forward_range R, class Proj = identity,}
\text{ indirect_unary_predicate<projected<iterator_t<R>, Proj>>, Pred>
\text{requires permutable<iterator_t<R>>,}
\text{ constexpr borrowed_subrange_t<R> ranges::\texttt{remove\_if}(R&& r, Pred pred, Proj proj = {});}

Let \( E \) be

(1.1) \( \text{bool(*i == value)} \) for \texttt{remove}:

(1.2) \( \text{bool(pred(*i))} \) for \texttt{remove\_if}:

(1.3) \( \text{bool(invoke(proj, *i) == value)} \) for \texttt{ranges::remove};

(1.4) \( \text{bool(invoke(pred, invoke(proj, *i)))} \) for \texttt{ranges::remove\_if}.

\textbf{Preconditions:} For the algorithms in namespace \texttt{std}, the type of \texttt{\*first} meets the \texttt{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:

- \( j \) for the overloads in namespace \( \text{std} \).
- \( \{j, \text{last}\} \) for the overloads in namespace \( \text{ranges} \).

Remarks: Stable (16.5.5.8).

Complexity: Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate and any projection.

[Note: Each element in the range \([\text{ret}, \text{last})\), where \( \text{ret} \) 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]

```cpp
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,
            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,
               ForwardIterator1 first, ForwardIterator1 last,
               ForwardIterator2 result, Predicate pred);

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class T,
           class Proj = identity>
requires indirectly_copyable<I, O> &&
indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>;
constexpr ranges::remove_copy_result<I, O>
ranges::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_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>;
constexpr ranges::remove_copy_result<borrowed_iterator_t<R>, O>
ranges::remove_copy(R&& r, O result, const T& value, Proj proj = {});

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 = {});

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 ranges::remove_copy_if_result<borrowed_iterator_t<R>, O>
ranges::remove_copy_if(R&& r, O result, Pred pred, Proj proj = {});

Let \( E \) be

- \( \text{bool}(*i == \text{value}) \) for \( \text{remove_copy} \);
- \( \text{bool}(\text{pred}(*i)) \) for \( \text{remove_copy_if} \);
- \( \text{bool}(\text{invoke}(\text{proj}, *i) == \text{value}) \) for \( \text{ranges::remove_copy} \);
```
Let $N$ be the number of elements in $[\text{first}, \text{last})$ for which $E$ is false.

Mandates: $\ast\text{first}$ 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: For the overloads with an ExecutionPolicy, there may 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.

{last, $\text{result} + N$}, for the algorithms in namespace ranges.

Complexity: Exactly $\text{last} - \text{first}$ applications of the corresponding predicate and any projection.

Remarks: Stable (16.5.5.8).

25.7.9 Unique

[alg.unique]

template<class ForwardIterator>
constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator unique(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
    BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    BinaryPredicate pred);

template<permutable I, sentinel_for<I> S, class Proj = identity,
    indirect_equivalence_relation<projected<I, Proj>> C = ranges::equal_to>
constexpr subrange<I> ranges::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>
ranges::unique(R&& r, C comp = {}, Proj proj = {});

Let $\text{pred}$ be equal_to{} for the overloads with no parameter $\text{pred}$, and let $E$ be

(1.1) $\text{bool}(\text{pred}(*(i - 1), *i))$ for the overloads in namespace std;
(1.2) $\text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *(i - 1)), \text{invoke}(\text{proj}, *i)))$ for the overloads in

name space ranges.

Preconditions: For the overloads in namespace std, $\text{pred}$ is an equivalence relation and the type of $\ast\text{first}$ meets the Cpp17MoveAssignible 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 $[\text{first} + 1, \text{last})$ for which $E$ is true.

Returns: Let $j$ be the end of the resulting range. Returns:

(4.1) $j$ for the overloads in namespace std.
(4.2) $\{j, \text{last}\}$ for the overloads in namespace ranges.

Complexity: For nonempty ranges, exactly $(\text{last} - \text{first}) - 1$ applications of the corresponding predicate and no more than twice as many applications of any projection.
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);  

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,
class BinaryPredicate>
ForwardIterator2
unique_copy(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryPredicate pred);  

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> ||
(indirectly_copyable_storable<I, O>))
constexpr ranges::unique_copy_result<I, O>
ranges::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>> ||
(indirectly_copyable_storable<iterator_t<R>, O>))
constexpr ranges::unique_copy_result<borrowed_iterator_t<R>, O>
ranges::unique_copy(R&& r, O result, C comp = {}, Proj proj = {});  

Let pred be equal_to{} for the overloads in namespace std with no parameter pred, and let E be
— bool(pred(*i, *(i - 1))) for the overloads in namespace std;
— 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:
— 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: For the overloads with an ExecutionPolicy, there may be a performance cost if the value type of ForwardIterator 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.

### 25.7.10 Reverse

[alg.reverse]

```cpp
template<class BidirectionalIterator>
constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator>
void reverse(ExecutionPolicy&& exec,
            BidirectionalIterator first, BidirectionalIterator last);
```

```cpp
template<bidirectional_iterator I, sentinel_for<I> S>
requires permutable<I>
constexpr I ranges::reverse(I first, S last);
```

```cpp
template<bidirectional_range R>
requires permutable<iterator_t<R>>
constexpr borrowed_iterator_t<R> ranges::reverse(R&& r);
```

1. **Preconditions:** For the overloads in namespace `std`, `BidirectionalIterator` meets the `Cpp17ValueSwappable` requirements (16.5.3.2).
2. **Effects:** For each non-negative integer `i < (last - first) / 2`, applies `std::iter_swap`, or `ranges::iter_swap` for the overloads in namespace `ranges`, to all pairs of iterators `first + i`, `(last - i) - 1`.
3. **Returns:** `last` for the overloads in namespace `ranges`.
4. **Complexity:** Exactly `(last - first)/2` swaps.

```cpp
template<class BidirectionalIterator, class OutputIterator>
constexpr OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                                      OutputIterator result);
```

```cpp
template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
ForwardIterator reverse_copy(ExecutionPolicy&& exec,
                              BidirectionalIterator first, BidirectionalIterator last,
                              ForwardIterator result);
```

```cpp
template<bidirectional_iterator I, sentinel_for<I> S, weakly_incrementable O>
requires indirectly_copyable<I, O>
constexpr ranges::reverse_copy_result<I, O>
ranges::reverse_copy(I first, S last, O result);
```

```cpp
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);
```

5. Let `N` be `last - first`.
6. **Preconditions:** The ranges `[first, last)` and `[result, result + N)` do not overlap.
7. **Effects:** Copies the range `[first, last)` to the range `[result, result + N)` such that for every non-negative integer `i < N` the following assignment takes place: `*(result + N - 1 - i) = *(first + i)`.
8. **Returns:**
   - `result + N` for the overloads in namespace `std`.
   - `{last, result + N}` for the overloads in namespace `ranges`.
9. **Complexity:** Exactly `N` assignments.
25.7.11 Rotate

[alg.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);

1 Preconditions: [first, middle) and [middle, last) are valid ranges. For the overloads in namespace std, ForwardIterator meets the Cpp17ValueSwappable requirements (16.5.3.2), and the type of *first meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

2 Effects: For each non-negative integer i < (last - first), places the element from the position first + i into position first + (i + (last - middle)) % (last - first). [Note: This is a left rotate. — end note]

3 Returns: (3.1) — first + (last - middle) for the overloads in namespace std.
(3.2) — {first + (last - middle), last} for the overload in namespace ranges.

4 Complexity: At most last - first swaps.

template<forward_range R>
requires permutable<iterator_t<R>>
constexpr borrowed_subrange_t<R>
ranges::rotate(R&& r, iterator_t<R> middle);

5 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>
requires indirectly_copyable<I, O>
constexpr ranges::rotate_copy_result<I, O>
ranges::rotate_copy(I first, I middle, S last, O result);

6 Let N be last - first.

7 Preconditions: [first, middle) and [middle, last) are valid ranges. The ranges [first, last) and [result, result + N) do not overlap.

8 Effects: Copies the range [first, last) to the range [result, result + N) such that for each non-negative integer i < N the following assignment takes place: *(result + i) = *(first + (i + (middle - first)) % N).

9 Returns: (9.1) — result + N for the overloads in namespace std.
(9.2) — {last, result + N} for the overload in namespace ranges.

10 Complexity: Exactly N assignments.

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:

return ranges::rotate_copy(ranges::begin(r), middle, ranges::end(r), result);

25.7.12 Sample

template<class PopulationIterator, class SampleIterator,
class Distance, class UniformRandomBitGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
SampleIterator out, Distance n,
UniformRandomBitGenerator&& g);

template<input_iterator I, sentinel_for<I> S, weakly_incrementable O, class Gen>
requires (forward_iterator<I> || random_access_iterator<O>) &&
indirectly_copyable<I, 0> &&
uniform_random_bit_generator<remove_reference_t<Gen>>
O ranges::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 ranges::sample(R&& r, O out, range_difference_t<R> n, Gen&& g);

Mandates: For the overload in namespace std, Distance is an integer type and *first is writable (23.3.1) to out.

Preconditions: out is not in the range [first, last). For the overload in namespace std:

1. PopulationIterator meets the Cpp17InputIterator requirements (23.3.5.2).
2. SampleIterator meets the Cpp17OutputIterator requirements (23.3.5.3).
3. SampleIterator meets the Cpp17RandomAccessIterator requirements (23.3.5.6) unless PopulationIterator meets the Cpp17ForwardIterator requirements (23.3.5.4).
4. remove_reference_t<UniformRandomBitGenerator> meets the requirements of a uniform random bit generator type (26.6.2.3).

Effects: Copies min(last - first, n) elements (the sample) from [first, last) (the population) to out such that each possible sample has equal probability of appearance. [Note: Algorithms that obtain such effects include selection sampling and reservoir sampling. — end note]

Returns: The end of the resulting sample range.

Complexity: \(O(\text{last} - \text{first})\).

Remarks:

1. 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.
2. 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

template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
UniformRandomBitGenerator&& g);

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);
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:
(1.1) — RandomAccessIterator meets the Cpp17ValueSwappable requirements (16.5.3.2).
(1.2) — The type remove_reference_t<UniformRandomBitGenerator> meets the uniform random bit generator (26.6.2.3) 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: Exactly (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

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);

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 + 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.

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 >= 0 is true. The type of *first meets the Cpp17MoveAssignable requirements.
ForwardIterator meets the Cpp17BidirectionalIterator requirements (23.3.5.5) or the Cpp17ValueSwappable requirements.

Effects: If n == 0 or n >= last - first, does nothing. Otherwise, moves the element from position first + i into position first + n + i for each non-negative integer i < (last - first) - n. In the first overload case, if ForwardIterator meets the Cpp17BidirectionalIterator requirements, does so in order starting from i = (last - first) - n - 1 and proceeding to i = 0.

Returns: first + n if n < last - first, otherwise last.

Complexity: At most (last - first) - n assignments or swaps.

25.8 Sorting and related operations

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 25.8.3, 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:

- (4.1) comp(a, b) && comp(b, c) implies comp(a, c)
- (4.2) equiv(a, b) && equiv(b, c) implies equiv(a, c)

[Note: Under these conditions, it can be shown that
- (4.3) equiv is an equivalence relation,
- (4.4) comp induces a well-defined relation on the equivalence classes determined by equiv, and
- (4.5) 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,

bool(invoke(comp, invoke(proj, *(i + n)), invoke(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 <= i < (finish - start), f(*(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).

25.8.1 Sorting [alg.sort]

25.8.1.1 sort [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 \code{less{}} and \(\text{proj}\) be \code{identity{}} for the overloads with no parameters by those names.

**Preconditions:** For the overloads in namespace \code{std}, \code{RandomAccessIterator} meets the \code{Cpp17ValueSwappable} requirements (16.5.3.2) and the type of \code{*first} meets the \code{Cpp17MoveConstructible} (Table 28) and \code{Cpp17MoveAssignable} (Table 30) requirements.

**Effects:** Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \(\text{comp}\) and \(\text{proj}\).

**Returns:** \(\text{last}\) for the overloads in namespace \code{ranges}.

**Complexity:** Let \(N\) be \(\text{last} - \text{first}\). \(\Theta(N \log N)\) comparisons and projections.

### 25.8.1.2 stable_sort

```cpp
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator>
void stable_sort(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void stable_sort(ExecutionPolicy&& exec,
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>
I ranges::stable_sort(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<random_access_range R, class Comp = ranges::less, class Proj = identity>
borrowed_iterator_t<R>
ranges::stable_sort(R&& r, Comp comp = {}, Proj proj = {});
```

Let \(\text{comp}\) be \code{less{}} and \(\text{proj}\) be \code{identity{}} for the overloads with no parameters by those names.

**Preconditions:** For the overloads in namespace \code{std}, \code{RandomAccessIterator} meets the \code{Cpp17ValueSwappable} requirements (16.5.3.2) and the type of \code{*first} meets the \code{Cpp17MoveConstructible} (Table 28) and \code{Cpp17MoveAssignable} (Table 30) requirements.

**Effects:** Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \(\text{comp}\) and \(\text{proj}\).

**Returns:** \(\text{last}\) for the overloads in namespace \code{ranges}.

**Complexity:** Let \(N\) be \(\text{last} - \text{first}\). If enough extra memory is available, \(N \log(N)\) comparisons. Otherwise, at most \(N \log_2^2(N)\) comparisons. In either case, twice as many projections as the number of comparisons.

**Remarks:** Stable (16.5.5.8).

### 25.8.1.3 partial_sort

```cpp
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);
```

Let \(\text{comp}\) be \code{less{}} and \(\text{proj}\) be \code{identity{}} for the overloads with no parameters by those names.

**Preconditions:** For the overloads in namespace \code{std}, \code{RandomAccessIterator} meets the \code{Cpp17ValueSwappable} requirements (16.5.3.2) and the type of \code{*first} meets the \code{Cpp17MoveConstructible} (Table 28) and \code{Cpp17MoveAssignable} (Table 30) requirements.

**Effects:** Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \(\text{comp}\) and \(\text{proj}\).

**Returns:** \(\text{last}\) for the overloads in namespace \code{ranges}.

**Complexity:** Let \(N\) be \(\text{last} - \text{first}\). 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 (16.5.5.8).
template<class RandomAccessIterator, class Compare>
constexpr void partial_sort(RandomAccessIterator first,
    RandomAccessIterator middle,
    RandomAccessIterator last,
    Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void partial_sort(ExecutionPolicy&& exec,
    RandomAccessIterator first,
    RandomAccessIterator middle,
    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::partial_sort(I first, I middle, S last, Comp comp = {}, Proj proj = {});

Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.

Preconditions: [first, middle) and [middle, last) are valid ranges. For the overloads in namespace std, RandomAccessIterator meets the Cpp17ValueSwappable requirements (16.5.3.2) and the type of *first meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

Effects: Places the first middle - first elements from the range [first, last) as sorted with respect to comp and proj into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

Returns: last for the overload in namespace ranges.

Complexity: Approximately (last - first) * log(middle - first) comparisons, and twice as many projections.

Preconditions: [first, middle) and [middle, last) are valid ranges. For the overloads in namespace std, RandomAccessIterator meets the Cpp17ValueSwappable requirements (16.5.3.2) and the type of *first meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) requirements.

Effects: Places the first middle - first elements from the range [first, last) as sorted with respect to comp and proj into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

Returns: last for the overload in namespace ranges.

Complexity: Approximately (last - first) * log(middle - first) comparisons, and twice as many projections.

Effects: Equivalent to:
return ranges::partial_sort(ranges::begin(r), middle, ranges::end(r), comp, proj);

25.8.1.4 partial_sort_copy

25.8.1.4.1097
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 comp be less{}, and proj1 and
proj2 be 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 Cpp17Value-
Swappable requirements (16.5.3.2), the type of *result_first meets the Cpp17MoveConstructible
(Table 28) and Cpp17MoveAssignable (Table 30) requirements.

**Preconditions:** 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: Writing a value from the input range into the output range does not affect how it is ordered by
comp and proj1 or proj2. — end note]

**Effects:** Places the first $N$ elements as sorted with respect to comp and proj2 into the range [result_-_first, result_first + $N$).

**Returns:**

(6.1) $\text{result_first + } N$ for the overloads in namespace std.

(6.2) $\text{\{last, result_first + } N$} for the overloads in namespace ranges.

**Complexity:** Approximately $(last - first) \times \log N$ comparisons, and twice as many projections.

**25.8.1.5 is_sorted**

```cpp
template<class ForwardIterator>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);
```

**Effects:** Equivalent to: return is_sorted_until(first, last) == last;

```cpp
template<class ExecutionPolicy, class ForwardIterator>
bool is_sorted(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last);
```

**Effects:** Equivalent to:

return is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last) == last;

§ 25.8.1.5
template<class ForwardIterator, class Compare>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
    Compare comp);

Effects: Equivalent to: return is_sorted_until(first, last, comp) == last;

template<class ExecutionPolicy, class ForwardIterator, class Compare>
bool is_sorted(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    Compare comp);

Effects: Equivalent to:
    return is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;

template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
    indirect_strict_weak_order<projected<I, Proj>> Comp = ranges::less>
constexpr bool ranges::is_sorted(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 bool ranges::is_sorted(R&& r, Comp comp = {}, Proj proj = {});

Effects: Equivalent to: return ranges::is_sorted_until(first, last, comp, proj) == last;

template<class ForwardIterator>
constexpr ForwardIterator
is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
is_sorted_until(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
constexpr ForwardIterator
is_sorted_until(ForwardIterator first, ForwardIterator last,
    Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator
is_sorted_until(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::is_sorted_until(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::is_sorted_until(R&& r, Comp comp = {}, Proj proj = {});

Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.

Returns: The last iterator i in [first, last] for which the range [first, i) is sorted with respect to comp and proj.

Complexity: Linear.

25.8.2 Nth element

[alg.nth.element]
template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
        class Proj = identity>
constexpr I
ranges::nth_element(I first, I nth, S last, Comp comp = {}, Proj proj = {});

1
Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.

2
Preconditions: [first, nth) and [nth, last) are valid ranges. For the overloads in namespace
std. RandomAccessIterator meets the Cpp17ValueSwappable requirements (16.5.3.2), and the type of
*first meets the Cpp17MoveConstructible (Table 28) and Cpp17MoveAssignable (Table 30) require-
ments.

3
Effects: After nth_element the element in the position pointed to by nth is the element that would be
in that position if the whole range were sorted with respect to comp and proj, unless nth == last.
Also for every iterator i in the range [first, nth) and every iterator j in the range [nth, last) it
holds that: bool(invoke(comp, invoke(proj, *j), invoke(proj, *i))) is false.

4
Returns: last for the overload in namespace ranges.

5
Complexity: For the overloads with no ExecutionPolicy, linear on average. For the overloads
with an ExecutionPolicy, \(\mathcal{O}(N)\) applications of the predicate, and \(\mathcal{O}(N \log N)\) swaps, where \(N = last - first\).

template<random_access_range R, class Comp = ranges::less, class Proj = identity>
constexpr borrowed_iterator_t<R>
ranges::nth_element(R&& r, iterator_t<R> nth, Comp comp = {}, Proj proj = {});

6
Effects: Equivalent to:
return ranges::nth_element(ranges::begin(r), nth, ranges::end(r), comp, proj);

25.8.3 Binary search

All of the algorithms in this subclause 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.

25.8.3.1 lower_bound

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);

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 = {});
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 = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

Preconditions: The elements e of [first, last) are partitioned with respect to the expression 
bool(invoke(comp, invoke(proj, e), value)).

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.

Complexity: At most log2(last - first) + Θ(1) comparisons and projections.

25.8.3.2 upper_bound

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);

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 = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

Preconditions: The elements e of [first, last) are partitioned with respect to the expression 
!bool(invoke(comp, value, invoke(proj, e))).

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.

Complexity: At most log2(last - first) + Θ(1) comparisons and projections.

25.8.3.3 equal_range

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 = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

Preconditions: The elements e of [first, last) are partitioned with respect to the expression 
!bool(invoke(comp, value, invoke(proj, e))).

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.

Complexity: At most log2(last - first) + Θ(1) comparisons and projections.
ranges::equal_range(R&& r, const T& value, Comp comp = {}, Proj proj = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

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.

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)}

Complexity: At most 2 * log₂(last - first) + Θ(1) comparisons and projections.

25.8.3.4 binary_search [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 = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

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.

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.

Complexity: At most log₂(last - first) + Θ(1) comparisons and projections.

25.8.4 Partitions [alg.partitions]

template<class InputIterator, class Predicate>
constexpr bool
is_partitioned(InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool
is_partitioned(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last, Predicate pred);

template<input_iterator I, sentinel_for<I> S, class Proj = identity,
indirect_unary_predicate<projected<I, Proj>> Pred>
constexpr bool ranges::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 ranges::is_partitioned(R&& r, Pred pred, Proj proj = {});

1 Let proj be identity{} for the overloads with no parameter named proj.
2 Returns: true if and only if the elements e of [first, last) are partitioned with respect to the
expression bool(invoke(pred, invoke(proj, e))).
3 Complexity: Linear. At most last - first applications of pred and proj.

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
partition(ForwardIterator first, ForwardIterator last, Predicate pred);

4 Let proj be identity{} for the overloads with no parameter named proj and let
E(x) = bool(invoke(pred, invoke(proj, x))).
5 Preconditions: For the overloads in namespace std, ForwardIterator meets the Cpp17ValueSwappable
requirements (16.5.3.2).
6 Effects: Places all the elements e in [first, last) that satisfy E(e) before all the elements that do
not.
7 Returns: Let i be an iterator such that E(*i) is true for every iterator j in [first, i) and false
for every iterator j in [i, last). Returns:
8 (7.1) — i for the overloads in namespace std.
8 (7.2) — {i, last} for the overloads in namespace ranges.
8 Complexity: Let N = last - first:
8 (8.1) — For the overload with no ExecutionPolicy, exactly N applications of the predicate and projection.
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.
8 (8.2) — For the overload with an ExecutionPolicy, O(N log N) swaps and O(N) applications of the
predicate.

§ 25.8.4 1103
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(involve(pred, invoke(proj, x))).

**Preconditions:** For the overloads in namespace std, BidirectionalIterator meets the Cpp17Value-Swappable requirements (16.5.3.2) 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:

- i for the overloads in namespace std.
- {i, last} for the overloads in namespace ranges.

**Complexity:** Let $N = last - first$:

- For the overloads with no ExecutionPolicy, at most $N \log N$ swaps, but only $O(N)$ swaps if there is enough extra memory. Exactly $N$ applications of the predicate and projection.

- For the overload with an ExecutionPolicy, $O(N \log N)$ swaps and $O(N)$ applications of the predicate.

§ 25.8.4 1104
Returns: Let $o_1$ be the end of the output range beginning at $\text{out\_true}$, and $o_2$ the end of the output range beginning at $\text{out\_false}$. Returns

- $\{o_1, o_2\}$ for the overloads in namespace std.
- $\{\text{last}, o_1, o_2\}$ for the overloads in namespace ranges.

Complexity: Exactly last - first applications of $\text{pred}$ and $\text{proj}$.

```
template<class ForwardIterator, class Predicate>
    constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);
```

```
template<class 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<class 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 = {});
```

```
Let $\text{proj}$ be identity{} for the overloads with no parameter named $\text{proj}$ and let $E(x)$ be bool(invoke($\text{pred}$, invoke($\text{proj}$, $x$))).
```

Preconditions: The elements $e$ of $[\text{first}, \text{last})$ are partitioned with respect to $E(e)$.

Returns: An iterator $\text{mid}$ such that $E(i)$ is true for all iterators $i$ in $[\text{first}, \text{mid})$, and false for all iterators $i$ in $[\text{mid}, \text{last})$.

Complexity: $O(\log(\text{last} - \text{first}))$ applications of $\text{pred}$ and $\text{proj}$.

### 25.8.5 Merge

```
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);
```

```
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 = {});
```
template<input_range R1, input_range R2, weakly_incrementable O, class Comp = ranges::less,
    class Proj1 = identity, class Proj2 = identity>
requires mergeable<input_iterator_t<R1>, input_iterator_t<R2>, O, Comp, Proj1, Proj2>
constexpr ranges::merge_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O>
ranges::merge(R1&& r1, R2&& r2, O result,
    Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

Let \( N = (last1 - first1) + (last2 - first2) \). Let \( \text{comp} = \text{less}{} \), \( \text{proj1} = \text{id}{} \), and \( \text{proj2} = \text{id}{} \), 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 \( result_last = result + N \). 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 \( \text{bool}(\text{invoke(\text{comp}, \text{invoke(\text{proj2}, e2), \text{invoke(\text{proj1}, e1)})}) = \text{true} \).

**Returns:**

(4.1) \( \text{result_last} \) for the overloads in namespace \text{std}.

(4.2) \( \{last1, last2, result_last\} \) for the overloads in namespace \text{ranges}.

**Complexity:**

(5.1) For the overloads with no \text{ExecutionPolicy}, at most \( N - 1 \) comparisons and applications of each projection.

(5.2) For the overloads with an \text{ExecutionPolicy}, \( \Theta(N) \) comparisons.

**Remarks:** Stable (16.5.5.8).

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 \( \text{comp} = \text{less}{} \) and \( \text{proj} = \text{id}{} \) for the overloads with no parameters by those names.

**Preconditions:** \([first, middle)\) and \([middle, last)\) are valid ranges sorted with respect to \( \text{comp} \) and \( \text{proj} \). For the overloads in namespace \text{std}, \text{BidirectionalIterator} meets the \text{Cpp17ValueSwappable} requirements (16.5.3.2) and the type of *\text{first} meets the \text{Cpp17MoveConstructible} (Table 28) and \text{Cpp17MoveAssignable} (Table 30) requirements.

**Effects:** Merges two sorted consecutive ranges \([first, middle)\) and \([middle, last)\), putting the result of the merge into the range \([first, last)\). The resulting range is sorted with respect to \( \text{comp} \) and \( \text{proj} \).
10 Returns: last for the overload in namespace ranges.

11 Complexity: Let \( N = \text{last} - \text{first} \):
11\(^{(1)}\) For the overloads with no ExecutionPolicy, and if enough additional memory is available, exactly \( N - 1 \) comparisons.
11\(^{(2)}\) Otherwise, \( \Theta(N \log N) \) comparisons.

In either case, twice as many projections as comparisons.

12 Remarks: Stable (16.5.5.8).

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 = {});

13 Effects: Equivalent to:

\[
\text{return ranges::inplace_merge(ranges::begin(r), middle, ranges::end(r), comp, proj);}\]

25.8.6 Set operations on sorted structures

This subclause 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 set_union to contain the maximum number of occurrences of every element, set_intersection to contain the minimum, and so on.

25.8.6.1 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<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 comp be less{}, proj1 be identity{}, and proj2 be identity{}, for the overloads with no parameters by those names.

1 Preconditions: The ranges [first1, last1] and [first2, last2] are sorted with respect to comp and proj1 or proj2, respectively.
Returns: true if and only if \([\text{first}_2, \text{last}_2)\) is a subsequence of \([\text{first}_1, \text{last}_1)\). [Note: A sequence \(S\) is a subsequence of another sequence \(T\) if \(S\) can be obtained from \(T\) by removing some, all, or none of \(T\)'s elements and keeping the remaining elements in the same order. — end note]

Complexity: At most \(2 \times ((\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)) - 1\) comparisons and applications of each projection.

25.8.6.2 set_union

(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 Comp = 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, 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<input_iterator_t<R1>, input_iterator_t<R2>, O, Comp, Proj1, Proj2>
constexpr ranges::set_union_result<input_iterator_t<R1>, borrowed_iterator_t<R2>, O> ranges::set_union(R1&& r1, R2&& r2, O result, Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

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: 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 \(\text{result}_\text{last}\) be the end of the constructed range. Returns

— \(\text{result}_\text{last}\) for the overloads in namespace std.

— \{\text{last}_1, \text{last}_2, \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: Stable (16.5.5.8). 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.

25.8.6.3 set_intersection

```
template<class InputIterator1, class InputIterator2,
    class OutputIterator>
constexpr OutputIterator
    set_intersection(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result);
```

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator>
ForwardIterator
    set_intersection(ExecutionPolicy&& exec,
                    ForwardIterator1 first1, ForwardIterator1 last1,
                    ForwardIterator2 first2, ForwardIterator2 last2,
                    ForwardIterator result);
```

```
template<class InputIterator1, class InputIterator2,
    class OutputIterator, class Compare>
constexpr OutputIterator
    set_intersection(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_intersection(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_intersection_result<I1, I2, O>
    ranges::set_intersection(I1 first1, S1 last1, I2 first2, S2 last2, 0 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_intersection_result<borrowed_iterator_t<R1>, borrowed_iterator_t<R2>, O>
    ranges::set_intersection(R1&& r1, R2&& r2, 0 result,
                             Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
```

Let `comp` be `less{}` and `proj1` and `proj2` be `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 `comp` and `proj1` or `proj2`, 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 `result_last` be the end of the constructed range. Returns

1. `result_last` for the overloads in namespace `std`.
2. `{\text{last}1, \text{last}2, \text{result_last}}` for the overloads in namespace `ranges`.
Complexity: At most \( 2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1 \) comparisons and applications of each projection.

Remarks: Stable (16.5.5.8). 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, the first \(%\min(m, n)\) elements are copied from the first range to the output range, in order.

25.8.6.4 set_difference

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator>
constexpr OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
ForwardIterator
set_difference(ExecutionPolicy&& exec,
               ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2,
               ForwardIterator result);
```

```cpp
template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
constexpr OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result, Compare comp);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
ForwardIterator
set_difference(ExecutionPolicy&& exec,
               ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, ForwardIterator2 last2,
               ForwardIterator result, Compare comp);
```

```cpp
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_difference_result<I1, O>
ranges::set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
                      Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
```

```cpp
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_difference_result<borrowed_iterator_t<R1>, O>
ranges::set_difference(R1&& r1, R2&& r2, O result,
                      Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
```

Let \(\text{comp}\) be \(\text{less}\{}\), and \(\text{proj1}\) and \(\text{proj2}\) be \(\text{identity}\{})\) for the overloads with no parameters by those names.

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.

Effects: Copies the elements of the range \([\text{first1}, \text{last1})\) which are not present in the range \([\text{first2}, \text{last2})\) to the range beginning at \(\text{result}\). The elements in the constructed range are sorted.

Returns: Let \(\text{result\_last}\) be the end of the constructed range. Returns

- \(\text{result\_last}\) for the overloads in namespace \text{std}.
- \{\text{last1}, \text{result\_last}\} for the overloads in namespace \text{ranges}.
Complexity: At most $2 \ast ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ comparisons and applications of each projection.

Remarks: 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, the last $\max(m - n, 0)$ elements from $[\text{first1}, \text{last1})$ is copied to the output range, in order.

25.8.6.5 `set_symmetric_difference`  

```
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 = {},  
Proj1 proj1 = {}, Proj2 proj2 = {});
```

Let comp be `less{}` and proj1 and proj2 be `identity{}` for the overloads with no parameters by those names.

Preconditions: The ranges $[\text{first1}, \text{last1})$ and $[\text{first2}, \text{last2})$ are sorted with respect to comp and proj1 or proj2, respectively. The resulting range does not overlap with either of the original ranges.

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 result. The elements in the constructed range are sorted.

Returns: Let `result_last` be the end of the constructed range. Returns $\text{-- result_last}$ for the overloads in namespace `std`. 

§ 25.8.6.5
— {last1, last2, result_last} for the overloads in namespace ranges.

**Complexity:** At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) comparisons and applications of each projection.

**Remarks:** Stable (16.5.5.8). If \([first1, last1)\) contains \(m\) elements that are equivalent to each other and \([first2, last2)\) 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 \([first1, last1)\) if \(m > n\), and the last \(n - m\) of these elements from \([first2, last2)\) if \(m < n\). In either case, the elements are copied in order.

### 25.8.7 Heap operations

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:

1. With \(N = b - a\), for all \(i, 0 < i < N\), bool(invoke(comp, invoke(proj, a\(\lfloor\frac{i-1}{2}\rfloor\))), invoke(proj, a\(i\))) is false.
2. \(*a\) may be removed by \(\text{pop_heap}\), or a new element added by \(\text{push_heap}\), in \(\Theta(\log N)\) time.

These properties make heaps useful as priority queues.

\(\text{make_heap}\) converts a range into a heap and \(\text{sort_heap}\) turns a heap into a sorted sequence.

#### 25.8.7.1 push_heap

**template<class RandomAccessIterator>**

```cpp
constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last);
```

**template<class RandomAccessIterator, class Compare>**

```cpp
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>**

```cpp
requires sortable<I, Comp, Proj>

constexpr I
ranges::push_heap(I first, S last, Comp comp = {}, Proj proj = {});
```

**template<random_access_range R, class Comp = ranges::less, class Proj = identity>**

```cpp
requires sortable<iterator_t<R>, Comp, Proj>

constexpr borrowed_iterator_t<R>
ranges::push_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 \([first, last - 1)\) is a valid heap with respect to \(\text{comp}\) and \(\text{proj}\). For the overloads in namespace \(\text{std}\), the type of \(*\text{first}\) meets the Cpp17MoveConstructible requirements (Table 28) and the Cpp17MoveAssignable requirements (Table 30).

**Effects:** Places the value in the location \(last - 1\) into the resulting heap \([first, last)\).

**Returns:** \(last\) for the overloads in namespace \(\text{ranges}\).

**Complexity:** At most \(\log(last - first)\) comparisons and twice as many projections.

#### 25.8.7.2 pop_heap

**template<class RandomAccessIterator>**

```cpp
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
```

**template<class RandomAccessIterator, class Compare>**

```cpp
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
                        Compare comp);
```

**template<random_access_iterator I, sentinel_for<I> S, class Comp = ranges::less,
class Proj = identity>**

```cpp
requires sortable<I, Comp, Proj>

constexpr I
ranges::pop_heap(I first, S last, Comp comp = {}, Proj proj = {});
```

§ 25.8.7.2
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 namespaces std, RandomAccessIterator meets the Cpp17ValueSwappable requirements (16.5.3.2) 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.

25.8.7.3 make_heap

[make.heap]
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); 

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 = {});

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.

25.8.7.4 sort_heap

[sort.heap]
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 comp be less{} and proj be 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 (16.5.3.2) and the type of *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.

25.8.7.5 \textbf{is_heap}

\[
\begin{align*}
\text{template} & \langle \text{class RandomAccessIterator} \rangle \\
& \text{constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);} \\
\text{Effects:} & \text{ Equivalent to: } \text{return is_heap_until(first, last) == last;} \\
\text{template} & \langle \text{class ExecutionPolicy, class RandomAccessIterator} \rangle \\
& \text{bool is_heap(ExecutionPolicy&& exec,} \\
& \quad \text{RandomAccessIterator first, RandomAccessIterator last);} \\
\text{Effects:} & \text{ Equivalent to: } \text{return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last) == last;} \\
\text{template} & \langle \text{class RandomAccessIterator, class Compare} \rangle \\
& \text{constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,} \\
& \quad \text{Compare comp);} \\
\text{Effects:} & \text{ Equivalent to: } \text{return is_heap_until(first, last, comp) == last;} \\
\text{template} & \langle \text{class ExecutionPolicy, class RandomAccessIterator, class Compare} \rangle \\
& \text{bool is_heap(ExecutionPolicy&& exec,} \\
& \quad \text{RandomAccessIterator first, RandomAccessIterator last,} \\
& \quad \text{Compare comp);} \\
\text{Effects:} & \text{ Equivalent to: } \text{return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;} \\
\end{align*}
\]

§ 25.8.7.5 1114
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 = {});

Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

Returns: The last iterator `i` in `[first, last]` for which the range `[first, i)` is a heap with respect to `comp` and `proj`.

Complexity: Linear.

### 25.8.8 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 = {});

Preconditions: For the first form, `T` meets the `Cpp17LessThanComparable` requirements (Table 26).

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent. An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

Complexity: Exactly one comparison and two applications of the projection, if any.

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 = {});

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> ranges::min(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 smallest value in the input range.

Remarks: Returns a copy of the leftmost element when several elements are equivalent to the smallest. An invocation may explicitly specify an argument for the template parameter `T` of the overloads in namespace `std`.

Complexity: Exactly `ranges::distance(r) - 1` comparisons and twice as many applications of the projection, if any.

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, 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.

Remarks: Returns the first argument when the arguments are equivalent. An invocation may explicitly
specify an argument for the template parameter T of the overloads in namespace std.

Complexity: Exactly one comparison and two applications of the projection, if any.

template<class T>
constexpr T max(initializer_list<T> r);

template<class T, class Compare>
constexpr T max(initializer_list<T> r, Compare comp);

template<copyable T, class Proj = identity,
         indirect_strict_weak_order<Projected<iterator_t<R>, Proj>> Comp = ranges::less>
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 Cpp17-
CopyConstructible requirements. For the first form, T meets the Cpp17LessThanComparable requirements
(Table 26).

Returns: The largest value in the input range.

Remarks: Returns a copy of the leftmost element when several elements are equivalent to the largest.
An invocation may explicitly specify an argument for the template parameter T of the overloads in
namespace std.

Complexity: Exactly ranges::distance(r) - 1 comparisons and twice as many applications of the
projection, if any.

° 25.8.8 1116
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>>

conestexpr ranges::minmax_result<range_value_t<R>>

ranges::minmax(R&& r, Comp 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.

Remarks: An invocation may explicitly specify an argument for the template parameter T of the overloads in namespace std.

Complexity: At most (3/2)ranges::distance(r) applications of the corresponding predicate and twice as many applications of the projection, if any.

template<class ForwardIterator>

conestexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>

ForwardIterator min_element(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>

conestexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last, Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class Compare>

ForwardIterator min_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<iterator_t<R>, Proj>> Comp = ranges::less>

conestexpr I ranges::min_element(I first, S last, Comp 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),

bool:invoke(comp, invoke(proj, *j), invoke(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>

conestexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>

ForwardIterator max_element(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>

conestexpr 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.

**25.8.9 Bounded value**

### [alg.clamp]

```cpp
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);

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 = {});
```

Let \( \text{comp} \) be \( \text{less}{} \) for the overloads with no parameter \( \text{comp} \), and let \( \text{proj} \) be \( \text{identity}{} \) for the overloads with no parameter \( \text{proj} \).

**Preconditions:** \( \text{bool}(\text{comp}(\text{proj}(\text{hi}), \text{proj}(\text{lo}))) \) is false. For the first form, type \( T \) meets the \textit{Cpp17LessThanComparable} requirements (Table 26).

---

234) This behavior intentionally differs from \textit{max_element}.
3 Returns:
lo if bool(comp(proj(v), proj(lo))) is true,
hi if bool(comp(proj(hi), proj(v))) is true,
otherwise v.  

[Note: If NaN is avoided, T can be a floating-point type. — end note]

Complexity: At most two comparisons and three applications of the projection.

25.8.10 Lexicographical 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<iterator_t<R1>, Proj1>,
iterator_t<R2>, Proj2>> Comp = ranges::less>
constexpr bool
ranges::lexicographical_compare(R1&& r1, R2&& r2,
Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

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).

Complexity: At most 2 \min(last2 - first1, last1 - first2) applications of the corresponding
comparison and each projection, if any.

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: ranges::lexicographical_compare(I1, S1, I2, S2, Comp, Proj1, Proj2) could be
implemented as:

\[
\text{for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) } \{
    \text{if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;}
    \text{if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;}
\}
\text{return first1 == last1 && first2 != last2;}
\]
— end example

[Note: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. — end note]

25.8.11 Three-way comparison algorithms

```
25.8.11 Three-way comparison algorithms \[alg.three.way\]

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));
```

`Mandates:` `decltype(comp(*b1, *b2))` is a comparison category type.

`Effects:` Lexicographically compares two ranges and produces a result of the strongest applicable comparison category type. Equivalent to:
```
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;
```

```
template<class InputIterator1, class InputIterator2>
constexpr auto
lexicographical_compare_three_way(InputIterator1 b1, InputIterator1 e1,
InputIterator2 b2, InputIterator2 e2);
```

`Effects:` Equivalent to:
```
return lexicographical_compare_three_way(b1, e1, b2, e2, compare_three_way());
```

25.8.12 Permutation generators

```
25.8.12 Permutation generators \[alg.permutation.generators\]
```

```
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);
```

```
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 = {});
```

```
template<bidirectional_range 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 = {});
```

1 Let `comp` be `less{}` and `proj` be `identity{}` for overloads with no parameters by those names.

2 Preconditions: For the overloads in namespace `std`, `BidirectionalIterator` meets the `Cpp17Value-Swappable` requirements (16.5.3.2).

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 \((\text{last} - \text{first}) / 2\) swaps.

```
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);
```

```
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 = {});
```

```
template<bidirectional_range 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 \(\text{comp}\) be \(\text{less}\{\}\) and \(\text{proj}\) be \(\text{identity}\{\}\) for overloads with no parameters by those names.

Preconditions: For the overloads in namespace \texttt{std}, \texttt{BidirectionalIterator} meets the \texttt{Cpp17Value-
Swappable} requirements (16.5.3.2).

Effects: Takes a sequence defined by the range \([\text{first}, \text{last})\) 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 \(\text{comp}\) and \(\text{proj}\). If no such permutation exists, transforms the
sequence into the last permutation; that is, the descendingly-sorted one.

Returns: Let \(\text{B}\) be \(\text{true}\) if a previous permutation was found and otherwise \(\text{false}\). Returns:

\[(9.1)\] \(\text{B}\) for the overloads in namespace \texttt{std}.

\[(9.2)\] \{ \text{last}, \text{B} \} for the overloads in namespace \texttt{ranges}.

Complexity: At most \((\text{last} - \text{first}) / 2\) swaps.

**25.9 Header <numeric> synopsis**

```
namespace std {
    // 25.10.2, 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);

    // 25.10.3, 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);
    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);
```

§ 25.9 1121
// 25.10.4, 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);

// 25.10.5, 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,
                   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,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, T init,
                   BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation, class UnaryOperation>
T transform_reduce(ExecutionPolicy&& exec,
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, T init,
                   BinaryOperation binary_op, UnaryOperation unary_op);

// 25.10.6, 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);

// 25.10.7, 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>
exclusive_scan(ExecutionPolicy&& exec,
               ForwardIterator1 first1, ForwardIterator1 last1,
               ForwardIterator2 first2, T init);
ForwardIterator2 result, T init);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
ForwardIterator2
exclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, T init, BinaryOperation binary_op);

// 25.10.8, 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 BinaryOperation>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryOperation binary_op, T init);

// 25.10.9, 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);

// 25.10.10, 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);

§ 25.9
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);

// 25.10.11, 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>
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);

// 25.10.12, iota
template<class ForwardIterator, class T>
constexpr void iota(ForwardIterator first, ForwardIterator last, T value);

// 25.10.13, greatest common divisor
template<class M, class N>
constexpr common_type_t<M,N> gcd(M m, N n);

// 25.10.14, least common multiple
template<class M, class N>
constexpr common_type_t<M,N> lcm(M m, N n);

// 25.10.15, midpoint
template<class T>
constexpr T midpoint(T a, T b) noexcept;
template<class T>
constexpr T* midpoint(T* a, T* b);

25.10 Generalized numeric operations

[numeric.ops]

1 [Note: The use of closed ranges as well as semi-open ranges to specify requirements throughout this subclause
is intentional. — end note]
25.10.1 Definitions

1 Define \( \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, a_1, \ldots, a_N) \) as follows:

\[
\begin{align*}
(1.1) & \quad \text{when } N = 1, \\
(1.2) & \quad \text{for any } K \text{ where } 1 < K + 1 = M \leq N.
\end{align*}
\]

2 Define \( \text{GENERALIZED\_SUM}(\text{op}, a_1, \ldots, a_N) \) as \( \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, b_1, \ldots, b_N) \), where \( b_1, \ldots, b_N \) may be any permutation of \( a_1, \ldots, a_N \).

25.10.2 Accumulate

\[
\text{template}<\text{class InputIterator, class T}>
\begin{align*}
\text{constexpr T accumulate}(\text{InputIterator first}, \text{InputIterator last}, \text{T init}); \\
\text{template}<\text{class InputIterator, class T, class BinaryOperation}>
\begin{align*}
\text{constexpr T accumulate}(\text{InputIterator first}, \text{InputIterator last}, \text{T init}, \text{BinaryOperation binary_op});
\end{align*}
\]

1 Preconditions: \( T \) meets the \text{Cpp17CopyConstructible} (Table 29) and \text{Cpp17CopyAssignable} (Table 31) requirements. In the range \([\text{first}, \text{last}]\), \text{binary_op} neither modifies elements nor invalidates iterators or subranges.

2 Effects: Computes its result by initializing the accumulator \( \text{acc} \) with the initial value \( \text{init} \) and then modifies it with \( \text{acc} = \text{std::move(acc)} + *i \) or \( \text{acc} = \text{binary_op(std::move(acc), *i)} \) for every iterator \( i \) in the range \([\text{first}, \text{last})\) in order.

25.10.3 Reduce

\[
\text{template}<\text{class InputIterator}>
\begin{align*}
\text{constexpr typename iterator_traits<InputIterator>::value_type reduce}(\text{InputIterator first}, \text{InputIterator last});
\end{align*}
\]

1 Effects: Equivalent to:

\[
\text{return reduce(first, last, typename iterator_traits<InputIterator>::value_type());}
\]

\[
\text{template}<\text{class ExecutionPolicy, class ForwardIterator}>
\begin{align*}
\text{typename iterator_traits<ForwardIterator>::value_type reduce}(\text{ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last});
\end{align*}
\]

2 Effects: Equivalent to:

\[
\text{return reduce(std::forward<ExecutionPolicy>(exec), first, last, typename iterator_traits<ForwardIterator>::value_type());}
\]

\[
\text{template}<\text{class InputIterator, class T}>
\begin{align*}
\text{constexpr T reduce}(\text{InputIterator first}, \text{InputIterator last}, \text{T init});
\end{align*}
\]

3 Effects: Equivalent to:

\[
\text{return reduce(first, last, init, plus<>());}
\]

\[
\text{template}<\text{class ExecutionPolicy, class ForwardIterator, class T}>
\begin{align*}
\text{T reduce}(\text{ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, T init});
\end{align*}
\]

4 Effects: Equivalent to:

\[
\text{return reduce(std::forward<ExecutionPolicy>(exec), first, last, init, plus<>());}
\]

(235) The use of fully closed ranges is intentional.
(236) \text{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.
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
(5.1) — binary_op(init, *first),
(5.2) — binary_op(*first, init),
(5.3) — binary_op(init, init), and
(5.4) — binary_op(*first, *first)
are convertible to T.

Preconditions:
(6.1) — T meets the Cpp17MoveConstructible (Table 28) requirements.
(6.2) — 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: 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]

25.10.4 Inner product [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);

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.237

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.

25.10.5 Transform reduce [transform.reduce]

template<class InputIterator1, class InputIterator2, class T>
constexpr T transform_reduce(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, T init);

Effects: Equivalent to:
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, ForwardIterator2 first2,

237) The use of fully closed ranges is intentional.
T init);

**Effects:** Equivalent to:

```cpp
return transform_reduce(std::forward<ExecutionPolicy>(exec),
  first1, last1, first2, init, plus<>(), multiplies<>());
```

```cpp
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);
```

```cpp
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

1. \( binary_{op1}(init, init) \)
2. \( binary_{op1}(init, binary_{op2}(*first1, *first2)) \)
3. \( binary_{op1}(binary_{op2}(*first1, *first2), init) \)
4. \( binary_{op1}(binary_{op2}(*first1, *first2), binary_{op2}(first1, *first2)) \)

are convertible to \( T \).

**Preconditions:**

1. \( T \) meets the `Cpp17MoveConstructible` (Table 28) requirements.
2. 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:** \( \Theta(last1 - first1) \) applications each of \( binary_{op1} \) and \( binary_{op2} \).

```cpp
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);
```

```cpp
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

1. \( binary_{op}(init, init) \)
2. \( binary_{op}(init, unary_{op}(*first)) \)
3. \( binary_{op}(unary_{op}(*first), init) \)
4. \( binary_{op}(unary_{op}(*first), unary_{op}(*first)) \)

are convertible to \( T \).

**Preconditions:**

§ 25.10.5
(8.1) - T meets the \textit{Cpp17MoveConstructible} (Table 28) requirements.
(8.2) - Neither \textit{unary\_op} nor \textit{binary\_op} invalidates subranges, nor modifies elements in the range [first, last].

9

\textbf{Returns:}

\textit{GENERALIZED\_SUM}(binary\_op, init, unary\_op(*i), ...)

for every iterator \(i\) in [first, last).

\textbf{Complexity:} \(\Theta(\text{last - first})\) applications each of \textit{unary\_op} and \textit{binary\_op}.

[\textit{Note: transform\_reduce} does not apply \textit{unary\_op} to \textit{init}. — \textit{end note}]

\section*{25.10.6 Partial sum [partial.sum]}

template<class InputIterator, class OutputIterator>
constexpr OutputIterator
\textbf{partial\_sum(InputIterator first, InputIterator last, OutputIterator result)};

template<class InputIterator, class OutputIterator, class BinaryOperation>
constexpr OutputIterator
\textbf{partial\_sum(InputIterator first, InputIterator last, OutputIterator result, BinaryOperation binary\_op)};

1

\textit{Mandates:} InputIterator's value type is constructible from *first. The result of the expression 
\texttt{std::move(acc) + *i} or \texttt{binary\_op(std::move(acc), *i)} is implicitly convertible to InputIterator's value type. acc is writable (23.3.1) to result.

2

\textit{Preconditions:} In the ranges [first, last] and [result, result + (last - first)] binary\_op neither modifies elements nor invalidates iterators or subranges.\textsuperscript{238}

3

\textit{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)).

4

\textit{Returns:} result + (last - first).

5

\textit{Complexity:} Exactly (last - first) - 1 applications of the binary operation.

6

\textit{Remarks:} result may be equal to first.

\section*{25.10.7 Exclusive scan [exclusive.scan]}

template<class InputIterator, class OutputIterator, class T>
constexpr OutputIterator
\textbf{exclusive\_scan(InputIterator first, InputIterator last, OutputIterator result, T init)};

1

\textit{Effects:} Equivalent to:

return exclusive\_scan(first, last, result, init, plus<>());

2

\textit{Effects:} Equivalent to:

return exclusive\_scan(std::forward<ExecutionPolicy>(exec), first, last, result, init, plus<>());

\textit{Remarks:} The use of fully closed ranges is intentional.
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
(3.1) — binary_op(init, init),
(3.2) — binary_op(init, *first), and
(3.3) — binary_op(*first, *first)
are convertible to T.

Preconditions:
(4.1) — T meets the Cpp17MoveConstructible (Table 28) requirements.
(4.2) — 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:
GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *(first + 0), *(first + 1), ..., *(first + K – 1))

Returns: The end of the resulting range beginning at result.

Complexity: \( \Theta((last - first)) \) applications of binary_op.

Remarks: result may be equal to first.

[Note: 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 may be nondeterministic. — end note]

25.10.8 Inclusive scan

template<class InputIterator, class OutputIterator>
constexpr OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
    OutputIterator result);

Effects: Equivalent to:
return inclusive_scan(first, last, result, plus<()());

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result);

Effects: Equivalent to:
return inclusive_scan(std::forward<ExecutionPolicy>(exec), first, last, result, plus<()());

template<class InputIterator, class OutputIterator, class BinaryOperation>
constexpr OutputIterator
inclusive_scan(InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryOperation>
ForwardIterator2
inclusive_scan(ExecutionPolicy&& exec,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 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, 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 `decltype(first)`.

**Mandates:** If `init` is provided, all of

1. $\text{binary}_\text{op}(\text{init}, \text{init})$,
2. $\text{binary}_\text{op}(\text{init}, *\text{first})$, and
3. $\text{binary}_\text{op}(*\text{first}, *\text{first})$

are convertible to $T$; otherwise, $\text{binary}_\text{op}(*\text{first}, *\text{first})$ is convertible to $U$.

**Preconditions:**

1. If `init` is provided, $T$ meets the `Cpp17MoveConstructible` (Table 28) requirements; otherwise, $U$ meets the `Cpp17MoveConstructible` requirements.
2. `$\text{binary}_\text{op}$` 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

1. $\text{GENERALIZED NONCOMMUTATIVE SUM}$
   
   $\text{binary}_\text{op}, \text{init}, *(first + 0), *(first + 1), \ldots, *(first + K)$
   
   if `init` is provided, or
2. $\text{GENERALIZED NONCOMMUTATIVE SUM}$
   
   $\text{binary}_\text{op}, *(first + 0), *(first + 1), \ldots, *(first + K)$
   
   otherwise.

**Returns:** The end of the resulting range beginning at `result`.

**Complexity:** $\mathcal{O}(last - first)$ applications of `binary_op`.

**Remarks:** `result` may be equal to `first`.

[Note: The difference between `exclusive_scan` and `inclusive_scan` is that `inclusive_scan` includes the $i$th input element in the $i$th sum. If `binary_op` is not mathematically associative, the behavior of `inclusive_scan` may be nondeterministic. — end note]

### 25.10.9 Transform exclusive scan

```cpp
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);
```

**Mandates:** All of

- $\text{binary}_\text{op}(\text{init}, \text{init})$,
- $\text{binary}_\text{op}(\text{init}, \text{unary}_\text{op}(*\text{first}))$, and
binary_op(unary_op(*first), unary_op(*first))

are convertible to T.

Preconditions:

T meets the Cpp17MoveConstructible (Table 28) requirements.

Neither unary_op nor binary_op invalidates iterators or subrange,

or 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:

\[
\text{GENERALIZED_NONCOMMUTATIVE_SUM} \\
\text{binary_op}, \text{init}, \\
\text{unary_op}((\text{first} + 0)), \text{unary_op}((\text{first} + 1)), \ldots, \text{unary_op}((\text{first} + K - 1))
\]

Returns: The end of the resulting range beginning at result.

Complexity: \(O(\text{last} - \text{first})\) applications each of unary_op and binary_op.

Remarks: result may be equal to first.

[Note: The difference between transform_exclusive_scan and 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 may be nondeterministic. transform_exclusive_scan does not apply unary_op to init. — end note]

25.10.10 Transform inclusive scan

[transform.inclusive.scan]

\begin{verbatim}
// 25.10.10 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 ExecutionPolicy, 
         class ForwardIterator1, class ForwardIterator2, 
         class BinaryOperation, class UnaryOperation>
ForwardIterator2 
transform_inclusive_scan(ExecutionPolicy&& exec, 
                        ForwardIterator1 first, ForwardIterator1 last, 
                        ForwardIterator2 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, class T>
ForwardIterator2 
transform_inclusive_scan(ExecutionPolicy&& exec, 
                        ForwardIterator1 first, ForwardIterator1 last, 
                        ForwardIterator2 result, 
                        BinaryOperation binary_op, UnaryOperation unary_op, 
                        T init);
\end{verbatim}

Let \(U\) be the value type of decltype(first).

Mandates: If init is provided, all of

\begin{itemize}
  \item binary_op(init, init),
  \item binary_op(init, unary_op(*first)), and
  \item binary_op(unary_op(*first), unary_op(*first))
\end{itemize}
are convertible to \( T \); otherwise, \( \text{binary\_op}(\text{unary\_op}(\ast \text{first}), \text{unary\_op}(\ast \text{first})) \) is convertible to \( U \).

**Preconditions:**

(3.1) If \( \text{init} \) is provided, \( T \) meets the \textit{Cpp17MoveConstructible} (Table 28) requirements; otherwise, \( U \) meets the \textit{Cpp17MoveConstructible} requirements.

(3.2) Neither \( \text{unary\_op} \) nor \( \text{binary\_op} \) invalidates iterators or subranges, nor modifies elements in the ranges \([\text{first}, \text{last}]\) or \([\text{result}, \text{result} + (\text{last} - \text{first})]\).

**Effects:** For each integer \( K \) in \([0, \text{last} - \text{first})\) assigns through \( \text{result} + K \) the value of

(4.1) \textit{GENERALIZED\_NONCOMMUTATIVE\_SUM}(
\begin{align*}
\text{binary\_op}, \text{init}, \\
\text{unary\_op}(\ast(\text{first} + 0)), \text{unary\_op}(\ast(\text{first} + 1)), \ldots, \text{unary\_op}(\ast(\text{first} + K))
\end{align*}
)
if \( \text{init} \) is provided, or

(4.2) \textit{GENERALIZED\_NONCOMMUTATIVE\_SUM}(
\begin{align*}
\text{binary\_op}, \\
\text{unary\_op}(\ast(\text{first} + 0)), \text{unary\_op}(\ast(\text{first} + 1)), \ldots, \text{unary\_op}(\ast(\text{first} + K))
\end{align*}
)
otherside.

**Returns:** The end of the resulting range beginning at \( \text{result} \).

**Complexity:** \( \Theta(\text{last} - \text{first}) \) applications each of \( \text{unary\_op} \) and \( \text{binary\_op} \).

**Remarks:** \( \text{result} \) may be equal to \( \text{first} \).

**[Note:** 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 \( \text{binary\_op} \) is not mathematically associative, the behavior of transform\_inclusive\_scan may be nondeterministic. transform\_inclusive\_scan does not apply \( \text{unary\_op} \) to \( \text{init} \). —end note]**

### 25.10.11 Adjacent difference

**[adjacent.difference]**

\[
\begin{align*}
\text{template<} &\text{class InputIterator, class OutputIterator}\nonumber \\
&\text{constexpr OutputIterator}\nonumber \\
&\text{adjacent\_difference(InputIterator first, InputIterator last, }\nonumber \\
&\text{OutputIterator result);}\nonumber \\
\text{template<} &\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2}\nonumber \\
&\text{ForwardIterator2}\nonumber \\
&\text{adjacent\_difference(ExecutionPolicy&& exec, }\nonumber \\
&\text{ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);}\nonumber \\
\text{template<} &\text{class InputIterator, class OutputIterator, class BinaryOperation}\nonumber \\
&\text{constexpr OutputIterator}\nonumber \\
&\text{adjacent\_difference(InputIterator first, InputIterator last, }\nonumber \\
&\text{OutputIterator result, BinaryOperation binary\_op);}\nonumber \\
\text{template<} &\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, }\nonumber \\
&\text{class BinaryOperation}\nonumber \\
&\text{ForwardIterator2}\nonumber \\
&\text{adjacent\_difference(ExecutionPolicy&& exec, }\nonumber \\
&\text{ForwardIterator1 first, ForwardIterator1 last, }\nonumber \\
&\text{ForwardIterator2 result, BinaryOperation binary\_op);}\nonumber \\
\end{align*}
\]

Let \( T \) be the value type of decltype(first). For the overloads that do not take an argument \( \text{binary\_op} \), let \( \text{binary\_op} \) be an lvalue that denotes an object of type minus<>.  

**Mandates:**

(2.1) For the overloads with no ExecutionPolicy, \( T \) is constructible from \(*\text{first}. \text{acc} \) (defined below) is writable (23.3.1) to the result output iterator. The result of the expression \( \text{binary\_op}(\text{val}, \text{std::move(}\text{acc})) \) is writable to result.

(2.2) For the overloads with an ExecutionPolicy, the result of the expressions \( \text{binary\_op}(\ast\text{first}, \ast\text{first}) \) and \( \ast\text{first} \) 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.\textsuperscript{239}

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.

25.10.12 Iota

```
template<class ForwardIterator, class T>
constexpr void iota(ForwardIterator first, ForwardIterator last, T value);
```

Mandates: `T` is convertible to `ForwardIterator`'s value type. The expression `++val`, where `val` has type `T`, is well-formed.

Effects: For each element referred to by the iterator `i` in the range `[first, last)`, assigns `*i = value` and increments `value` as if by `++value`.

Complexity: Exactly `last - first` increments and assignments.

25.10.13 Greatest common divisor

```
template<class M, class N>
constexpr common_type_t<M,N> gcd(M m, N n);
```

Mandates: `M` and `N` both are integer types and not `cv` `bool`.

Preconditions: `|m|` and `|n|` are representable as a value of `common_type_t<M, N>`.

Note: 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.

25.10.14 Least common multiple

```
template<class M, class N>
constexpr common_type_t<M,N> lcm(M m, N n);
```

Mandates: `M` and `N` both are integer types and not `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.

\textsuperscript{239} The use of fully closed ranges is intentional.
25.10.15 Midpoint

```
template<class T>
constexpr T midpoint(T a, T b) noexcept;
```

1 Constraints: T is an arithmetic type other than bool.
2 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.
3 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);
```

4 Constraints: T is an object type.
5 Mandates: T is a complete type.
6 Preconditions: a and b point to, respectively, elements i and j of the same array object x. [Note: As specified in 6.8.2, 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]
7 Returns: A pointer to array element \( i + \frac{j}{2} \) of x, where the result of the division is truncated towards zero.

25.11 Specialized <memory> algorithms

The contents specified in this subclause 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 (7.6.2.7) are destroyed in an unspecified order before allowing the exception to propagate.

[Note: When invoked on ranges of potentially-overlapping subobjects (6.7.2), the algorithms specified in this subclause 25.11 result in undefined behavior. —end note]

Some algorithms specified in this clause make use of the exposition-only function `voidify`:

```
template<class T>
constexpr void* voidify(T& obj) noexcept {
    return const_cast<void*>(static_cast<const volatile void*>(addressof(obj)));
}
```

25.11.1 Special memory concepts

Some algorithms in this subclause are constrained with the following exposition-only concepts:

```
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: This concept allows some `input_iterator` (23.3.4.9) operations to throw exceptions. —end note]

```
template<class S, class I>
concept no-throw-sentinel = sentinel_for<S, I>; // exposition only
```

Types S and I model `no-throw-sentinel` 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: 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<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<I, I>;

[Note: 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>>;

25.11.2 uninitialized_default_construct

    template<class NoThrowForwardIterator>
    void uninitialized_default_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<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_t<R> uninitialized_default_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_default_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;
            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);
    }

4 Effects: Equivalent to:
            return uninitialized_default_construct(counted_iterator(first, n),
                default_sentinel).base();
25.11.3 uninitialized_value_construct

```cpp
template<class NoThrowForwardIterator>
void uninitialized_value_construct(NoThrowForwardIterator first, NoThrowForwardIterator last);
```

1 Effects: Equivalent to:
   ```cpp
   for (; first != last; ++first)
       ::new (voidify(*first))
       typename iterator_traits<NoThrowForwardIterator>::value_type();
   ```

namespace ranges {

```cpp
template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
I uninitialized_value_construct(I first, S last);
```

2 Effects: Equivalent to:
   ```cpp
   for (; first != last; ++first)
       ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>();
   return first;
   ```

```cpp
template<class NoThrowForwardIterator, class Size>
NoThrowForwardIterator uninitialized_value_construct_n(NoThrowForwardIterator first, Size n);
```

3 Effects: Equivalent to:
   ```cpp
   for (; n > 0; (void)++first, --n)
       ::new (voidify(*first))
       typename iterator_traits<NoThrowForwardIterator>::value_type();
   return first;
   ```

namespace ranges {

```cpp
template<no-throw-forward-iterator I>
I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
```

4 Effects: Equivalent to:
   ```cpp
   return uninitialized_value_construct(counted_iterator(first, n),
                                       default_sentinel).base();
   ```

25.11.4 uninitialized_copy

```cpp
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:
   ```cpp
   for (; first != last; ++result, (void) ++first)
       ::new (voidify(*result))
       typename iterator_traits<NoThrowForwardIterator>::value_type(*first);
   ```

3 Returns: result.

namespace ranges {

```cpp
template<input_iterator I, sentinel_for<I> S1,
           no-throw-forward-iterator O, no-throw-sentinel<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(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<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);
}

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};

25.11.5 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<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, 0 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);
Preconditions: \([\text{ofirst, olast})\) does not overlap with \([\text{ifirst, ilast})\).

Effects: Equivalent to:

```cpp
for (; \text{ifirst} != \text{ilast} \&\& \text{ofirst} != \text{olast}; ++\text{ofirst}, (void)++\text{ifirst}) {
    \text{::new (voidify(*ofirst))}
    \text{remove_reference_t<iter_reference_t<O>>(ranges::iter_move(\text{ifirst}))};
} return \{\text{std::move(\text{ifirst}), ofirst}\};
```

[Note: If an exception is thrown, some objects in the range \([\text{first, last})\) 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: \(\text{result} + [0, n)\) does not overlap with \(\text{first} + [0, n)\).

Effects: Equivalent to:

```cpp
for (; n > 0; ++\text{result}, (void) ++\text{first}, --n) :
    \text{::new (voidify(*\text{result}))}
    \text{typename iterator_traits<NoThrowForwardIterator>::value_type(std::move(*\text{first}))};
return \{\text{first, result}\};
```

namespace ranges {
    template<input_iterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
    requires constructible_from<iter_value_t<O>, iter_rvalue_reference_t<I>>
    uninitialized_move_n_result<I, O>
    uninitialized_move_n(I \text{ifirst}, iter_difference_t<I> n, 0 ofirst, S \text{olast});
}

Preconditions: \([\text{ofirst, olast})\) does not overlap with \(\text{ifirst} + [0, n)\).

Effects: Equivalent to:

```cpp
auto t = uninitialized_move(counted_iterator(\text{ifirst}, n),
    default_sentinel, ofirst, olast);
return \{\text{std::move(t.in).base()}, t.out\};
```

[Note: If an exception is thrown, some objects in the range \(\text{first} + [0, n)\) are left in a valid but unspecified state. — end note]

25.11.6 uninitialized_fill

```
template<class NoThrowForwardIterator, class T>
void uninitialized_fill(NoThrowForwardIterator first, NoThrowForwardIterator last, const T& x);  
```

Effects: Equivalent to:

```cpp
for (; first != last; ++\text{first}) :
    \text{::new (voidify(*\text{first}))}
    \text{typename iterator_traits<NoThrowForwardIterator>::value_type(x)};
```

namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
    requires constructible_from<iter_value_t<I>, const T&>
    I uninitialized_fill(I \text{first}, S \text{last}, const T& \text{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\& \text{r}, const T& \text{x});
}

Effects: Equivalent to:

```cpp
for (; first != last; ++\text{first}) {
    \text{::new (voidify(*\text{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);

2  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);
}

3  Effects: Equivalent to:
    return uninitialized_fill(counted_iterator(first, n), default_sentinel, x).base();

25.11.7  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);
}

1  Constraints: The expression ::new (declval<void*>()) T(declval<Args>()...) is well-formed
      when treated as an unevaluated operand.

2  Effects: Equivalent to:
    return ::new (voidify(*location)) T(std::forward<Args>(args)...);

25.11.8  destroy  [specialized.destroy]

template<class T>
constexpr void destroy_at(T* location);

namespace ranges {
  template<destructible T>
  constexpr void destroy_at(T* location) noexcept;
}

1  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);

2  Effects: Equivalent to:
    for (; first != last; ++first)
      destroy_at(addressof(*first));

namespace ranges {
  template<no-throw-input-iterator I, no-throw-sentinel<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;
}

3  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:
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]

[Note: 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);

Effects: These functions have the semantics specified in the C standard library.

Preconditions: The objects in the array pointed to by base are of trivial type.

Throws: Any exception thrown by compar (16.5.5.13).

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>Subclause Header</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.2 Requirements</td>
<td></td>
</tr>
<tr>
<td>26.3 Floating-point environment</td>
<td>&lt;cfenv&gt;</td>
</tr>
<tr>
<td>26.4 Complex numbers</td>
<td>&lt;complex&gt;</td>
</tr>
<tr>
<td>26.5 Bit manipulation</td>
<td>&lt;bit&gt;</td>
</tr>
<tr>
<td>26.6 Random number generation</td>
<td>&lt;random&gt;</td>
</tr>
<tr>
<td>26.7 Numeric arrays</td>
<td>&lt;valarray&gt;</td>
</tr>
<tr>
<td>26.8 Mathematical functions for floating-point types</td>
<td>&lt;cmath&gt;, &lt;cstdlib&gt;</td>
</tr>
<tr>
<td>26.9 Numbers</td>
<td>&lt;numbers&gt;</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 (16.5.3.1).

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: 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> synopsis

#define FE_ALL_EXCEPT see below
#define FE_DIVBYZERO 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    // optional

240) In other words, value types. These include arithmetic types, pointers, the library class complex, and instantiations of valarray for value types.

§ 26.3.1
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: 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]

2 The floating-point environment has thread storage duration (6.7.5.2). 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 (32.4.2) or jthread object (32.4.3) at the time it constructed the object. [Note: That is, the child thread gets the floating-point state of the parent thread at the time of the child’s creation.
—end note]

3 A separate floating-point environment is maintained for each thread. Each function accesses the environment
corresponding to its calling thread.

See also: ISO C 7.6

26.4 Complex numbers [complex.numbers]

1 The header `<complex>` defines a class template, and numerous functions for representing and manipulating
complex numbers.

2 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).

3 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.

4 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].
26.4.1 Header <complex> synopsis

namespace std {
    // 26.4.2, class template complex
    template<class T> class complex;

    // 26.4.3, specializations
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;

    // 26.4.6, 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> constexpr bool operator==(const complex<T>&, const complex<T>&);
    template<class T> constexpr bool operator==(const complex<T>&, const 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>&);

    // 26.4.7, 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> 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());

    // 26.4.8, transcendentals
    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>&);

§ 26.4.1 1143
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>&);  

// 26.4.10, 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);  
  }
}

26.4.2 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>&);  
  };
}

The class complex describes an object that can store the Cartesian components, real() and imag(), of a complex number.

26.4.3 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 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 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 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 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 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>&);

26.4.4 Member functions

template<class T> constexpr complex(const T& re = T(), const T& im = T());

Postconditions: real() == re && imag() == im is true.

constexpr T real() const;
Returns: The value of the real component.

constexpr void real(T val);
Effects: Assigns val to the real component.

constexpr T imag() const;
Returns: The value of the imaginary component.

constexpr void imag(T val);
Effects: Assigns val to the imaginary component.

26.4.5 Member operators

constexpr complex& operator+=(const T& rhs);
Effects: Adds the scalar value 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.
Returns: *this.

constexpr complex& operator-=(const T& rhs);
Effects: Subtracts the scalar value 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.
Returns: *this.

constexpr complex& operator*=(const T& rhs);
Effects: Multiplies the scalar value rhs by the complex value *this and stores the result in *this.
Returns: *this.

constexpr complex& operator/=(const T& rhs);
Effects: Divides the scalar value rhs into the complex value *this and stores the result in *this.
Returns: *this.

template<class X> constexpr complex& operator+=(const complex<X>& rhs);
Effects: Adds the complex value rhs to the complex value *this and stores the sum in *this.
Returns: *this.
template<class X> constexpr complex& operator-=(const complex<X>& rhs);

Effects: Subtracts the complex value rhs from the complex value *this and stores the difference in *this.
Returns: *this.

template<class X> constexpr complex& operator*=(const complex<X>& rhs);

Effects: Multiplies the complex value rhs by the complex value *this and stores the product in *this.
Returns: *this.

template<class X> constexpr complex& operator/=(const complex<X>& rhs);

Effects: Divides the complex value rhs into the complex value *this and stores the quotient in *this.
Returns: *this.

26.4.6 Non-member operations

template<class T> constexpr complex<T> operator+(const complex<T>& lhs);

Returns: 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: complex<T>(lhs) += 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: complex<T>(lhs) -= 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: complex<T>(lhs) *= 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: complex<T>(lhs) /= 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: lhs.real() == rhs.real() && lhs.imag() == rhs.imag().

Remarks: The imaginary part is assumed to be T(), or 0.0, for the T arguments.

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 (29.7.4.2).
If bad input is encountered, calls is.setstate(ios_base::failbit) (which may throw ios_base::failure (29.5.5.4)).

Returns: 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.
template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);

Effects: Inserts the complex number \( x \) onto the stream \( o \) as if it were implemented as follows:

```cpp
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();
```

[Note: 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]

### 26.4.7 Value operations

- **template<class T> constexpr T real(const complex<T>& x);**
  
  Returns: \( x.\text{real()} \).

- **template<class T> constexpr T imag(const complex<T>& x);**
  
  Returns: \( x.\text{imag()} \).

- **template<class T> T abs(const complex<T>& x);**
  
  Returns: The magnitude of \( x \).

- **template<class T> T arg(const complex<T>& x);**
  
  Returns: The phase angle of \( x \), or \( \text{atan2}(\text{imag}(x), \text{real}(x)) \).

- **template<class T> constexpr T norm(const complex<T>& x);**
  
  Returns: The squared magnitude of \( x \).

- **template<class T> complex<T> conj(const complex<T>& x);**
  
  Returns: The complex conjugate of \( x \).

- **template<class T> complex<T> proj(const complex<T>& x);**
  
  Returns: The projection of \( x \) onto the Riemann sphere.

  **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());**
  
  **Preconditions:** \( \text{rho} \) is non-negative and non-NaN. \( \text{theta} \) is finite.

  Returns: The complex value corresponding to a complex number whose magnitude is \( \text{rho} \) and whose phase angle is \( \text{theta} \).

### 26.4.8 Transcendentals

- **template<class T> complex<T> acos(const complex<T>& x);**
  
  Returns: The complex arc cosine of \( x \).

  **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);**
  
  Returns: The complex arc sine of \( x \).

  **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);**
  
  Returns: The complex arc tangent of \( x \).

  **Remarks:** Behaves the same as the C function catan. See also: ISO C 7.3.5.3
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 \texttt{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 \texttt{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 \texttt{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 \([-\pi, \pi]\). [\text{Note}: The semantics of this function are intended to be the same in C++ as they are for \texttt{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. [\text{Note}: The semantics of this function are intended to be the same in C++ as they are for \texttt{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 \( x \).
26.4.9 Additional overloads
1 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>.
3 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>.

26.4.10 Suffixes for complex number literals
1 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
1 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;
   }
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 (6.8.1).
2    Returns: true if x is an integral power of two; false otherwise.
Let $N$ be the smallest power of 2 greater than or equal to $x$.

**Constraints:** $T$ is an unsigned integer type (6.8.1).

**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).

**Constraints:** $T$ is an unsigned integer type (6.8.1).

**Returns:** If $x == 0$, 0; otherwise the maximal value $y$ such that $\text{has\_single\_bit}(y)$ is true and $y <= x$.

**Constraints:** $T$ is an unsigned integer type (6.8.1).

**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`.

**template<class T>**

`[[nodiscard]] constexpr T rotl(T x, int s) noexcept;`

**Constraints:** $T$ is an unsigned integer type (6.8.1).

Let $r$ be $s \% N$.

**Returns:** If $r$ is 0, $x$; if $r$ is positive, $(x << r) | (x >> (N - r))$; if $r$ is negative, $\text{rotr}(x, -r)$.

**template<class T>**

`[[nodiscard]] constexpr T rotr(T x, int s) noexcept;`

**Constraints:** $T$ is an unsigned integer type (6.8.1).

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`.

**template<class T>**

`constexpr int countl_zero(T x) noexcept;`

**Constraints:** $T$ is an unsigned integer type (6.8.1).

**Returns:** The number of consecutive 0 bits in the value of $x$, starting from the most significant bit.

[Note: Returns $N$ if $x == 0$. — end note]

**template<class T>**

`constexpr int countl_one(T x) noexcept;`

**Constraints:** $T$ is an unsigned integer type (6.8.1).

**Returns:** The number of consecutive 1 bits in the value of $x$, starting from the most significant bit.

[Note: Returns $N$ if $x == \text{numeric\_limits<T>::max()}$. — end note]

**template<class T>**

`constexpr int countr_zero(T x) noexcept;`

**Constraints:** $T$ is an unsigned integer type (6.8.1).
6 Returns: The number of consecutive 0 bits in the value of x, starting from the least significant bit.
[Note: Returns N if x == 0. — end note]

template<class T>
constexpr int countr_one(T x) noexcept;

7 Constraints: T is an unsigned integer type (6.8.1).

8 Returns: The number of consecutive 1 bits in the value of x, starting from the least significant bit.
[Note: Returns N if x == numeric_limits<T>::max(). — end note]

template<class T>
constexpr int popcount(T x) noexcept;

9 Constraints: T is an unsigned integer type (6.8.1).

10 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
};

2 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

This subclause defines a facility for generating (pseudo-)random numbers.

2 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: 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]

3 Each of the entities specified via this subclause has an associated arithmetic type (6.8.1) 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.

4 Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

5 Throughout this subclause, 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 lshiftw denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo \(2^w\).
26.6.1 Header `<random>` synopsis

```
#include <initializer_list>

namespace std {
    // 26.6.3.1, class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

    // 26.6.3.2, 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;

    // 26.6.3.3, class template subtract_with_carry_engine
    template<class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

    // 26.6.4.2, class template discard_block_engine
    template<class Engine, size_t p, size_t r>
    class discard_block_engine;

    // 26.6.4.3, class template independent_bits_engine
    template<class Engine, size_t w, class UIntType>
    class independent_bits_engine;

    // 26.6.4.4, class template shuffle_order_engine
    template<class Engine, size_t k>
    class shuffle_order_engine;

    // 26.6.5, 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 ranlux24_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;

    // 26.6.6, class random_device
    class random_device;

    // 26.6.7.1, class seed_seq
    class seed_seq;

    // 26.6.7.2, function template generate_canonical
    template<class RealType, size_t bits, class URBG>
    RealType generate_canonical(URBG& g);

    // 26.6.8.2.1, class template uniform_int_distribution
    template<class IntType = int>
    class uniform_int_distribution;
```
// 26.6.8.2.2, class template uniform_real_distribution
template<class RealType = double>
class uniform_real_distribution;

// 26.6.8.3.1, class bernoulli_distribution
class bernoulli_distribution;

// 26.6.8.3.2, class template binomial_distribution
template<class IntType = int>
class binomial_distribution;

// 26.6.8.3.3, class template geometric_distribution
template<class IntType = int>
class geometric_distribution;

// 26.6.8.3.4, class template negative_binomial_distribution
template<class IntType = int>
class negative_binomial_distribution;

// 26.6.8.4.1, class template poisson_distribution
template<class IntType = int>
class poisson_distribution;

// 26.6.8.4.2, class template exponential_distribution
template<class RealType = double>
class exponential_distribution;

// 26.6.8.4.3, class template gamma_distribution
template<class RealType = double>
class gamma_distribution;

// 26.6.8.4.4, class template weibull_distribution
template<class RealType = double>
class weibull_distribution;

// 26.6.8.4.5, class template extreme_value_distribution
template<class RealType = double>
class extreme_value_distribution;

// 26.6.8.5.1, class template normal_distribution
template<class RealType = double>
class normal_distribution;

// 26.6.8.5.2, class template lognormal_distribution
template<class RealType = double>
class lognormal_distribution;

// 26.6.8.5.3, class template chi_squared_distribution
template<class RealType = double>
class chi_squared_distribution;

// 26.6.8.5.4, class template cauchy_distribution
template<class RealType = double>
class cauchy_distribution;

// 26.6.8.5.5, class template fisher_f_distribution
template<class RealType = double>
class fisher_f_distribution;

// 26.6.8.5.6, class template student_t_distribution
template<class RealType = double>
class student_t_distribution;


```cpp
// 26.6.8.6.1, class template discrete_distribution
template<class IntType = int>
  class discrete_distribution;

// 26.6.8.6.2, class template piecewise_constant_distribution
template<class RealType = double>
  class piecewise_constant_distribution;

// 26.6.8.6.3, class template piecewise_linear_distribution
template<class RealType = double>
  class piecewise_linear_distribution;
```

26.6.2 Requirements

26.6.2.1 General requirements

(1.1) — that has a template type parameter named $S_{seq}$ is undefined unless the corresponding template argument is cv-unqualified and meets the requirements of seed sequence (26.6.2.2).

(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 (26.6.2.3).

(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 (26.6.2.4).

(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.

26.6.2.2 Seed sequence requirements

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: 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]

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 26.6.2.2. 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 (6.8.1) 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>same as S(il.begin(), il.end())</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>

26.6.2.3 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: The degree to which $g$’s results approximate the ideal is often determined statistically. — end note]

\[
\text{template<class G>}
\text{
  concept uniform_random_bit_generator =}
\text{
  invocable<G&> && unsigned_integral<invoke_result_t<G&>> &&}
\text{
  requires {
    \{ G::min() \} -> same_as<invoke_result_t<G&>>;}
\text{
    \{ G::max() \} -> same_as<invoke_result_t<G&>>;}
\text{
    \{ bool\_constant<(G::min() \lt G::max())>::value; \};}
\]

2 Let $g$ be an object of type $G$. $G$ models uniform_random_bit_generator only if

\[ (2.1) \quad \text{-- } G::\text{min() } \leq g(), \]

\[ (2.2) \quad \text{-- } g() \leq G::\text{max()}, \text{ and } \]

\[ (2.3) \quad \text{-- } g() \text{ has amortized constant complexity.} \]

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 (6.8.1), and $G$ provides a nested typedef-name result_type that denotes the same type as invoke_result_t<G&>.
26.6.2.4 Random number engine requirements [rand.req.eng]

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 (26.6.2.3) 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 26.6.2.4. 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 (26.6.2.2);
- \( z \) is a value of type unsigned long long;
- \( os \) is an lvalue of the type of some class template specialization basic_ostream<charT, traits>;
- \( is \) 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 [tab:rand.req.eng]

<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>( \mathcal{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>( \mathcal{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>( \mathcal{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() ). same as ( E() )</td>
<td></td>
</tr>
<tr>
<td>( e.seed(s) )</td>
<td>void</td>
<td>Postconditions: ( e == E(s) ). same as ( E(s) )</td>
<td></td>
</tr>
<tr>
<td>( e.seed(q) )</td>
<td>void</td>
<td>Postconditions: ( e == E(q) ). same as ( E(q) )</td>
<td></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 26.6.2.3</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() ). no worse than the complexity of ( z ) consecutive calls ( e() )</td>
<td></td>
</tr>
</tbody>
</table>

---

241) This constructor (as well as the subsequent corresponding seed() function) may be particularly useful to applications requiring a large number of independent random sequences.

242) 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 \( e() \).

§ 26.6.2.4 1158
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == y</code></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 <code>x()</code> and <code>y()</code>, respectively, returns <code>true</code> if $S_x = S_y$; else returns <code>false</code>.</td>
<td>$O$(size of state)</td>
</tr>
<tr>
<td><code>x != y</code></td>
<td>bool</td>
<td><code>!(x == y)</code></td>
<td>$O$(size of state)</td>
</tr>
<tr>
<td><code>os &lt;&lt; x</code></td>
<td>reference to the type of <code>os</code></td>
<td>With <code>os.fmtflags</code> set to `ios_base::dec</td>
<td>ios_base::left<code>and the fill character set to the space character, writes to</code>os<code>the textual representation of</code>x<code>'s current state. In the output, adjacent numbers are separated by one or more space characters. **Postconditions:** The </code>os.fmtflags` and fill character are unchanged.</td>
</tr>
<tr>
<td><code>is &gt;&gt; v</code></td>
<td>reference to the type of <code>is</code></td>
<td>With <code>is.fmtflags</code> set to <code>ios_base::dec</code>, sets <code>v</code>'s state as determined by reading its textual representation from <code>is</code>. If bad input is encountered, ensures that <code>v</code>'s state 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>). If a textual representation written via <code>os &lt;&lt; x</code> was subsequently read via <code>is &gt;&gt; v</code>, then <code>x == v</code> provided that there have been no intervening invocations of <code>x</code> or of <code>v</code>. <strong>Preconditions:</strong> <code>is</code> provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of <code>is</code>, and whose type’s template specialization arguments <code>charT</code> and <code>traits</code> were respectively the same as those of <code>is</code>. <strong>Postconditions:</strong> The <code>is.fmtflags</code> are unchanged.</td>
<td>$O$(size of state)</td>
</tr>
</tbody>
</table>

5 E shall meet the Cpp17CopyConstructible (Table 29) and Cpp17CopyAssignble (Table 31) requirements. These operations shall each be of complexity no worse than $O$(size of state).

26.6.2.5 Random number engine adaptor requirements

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.base()` 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.

A::A();

Effects: The base engine is initialized as if by its default constructor.

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.

A::A(result_type s);

Effects: The base engine is initialized with s.

template<class Sseq> A::A(Sseq& q);

Effects: The base engine is initialized with q.

void seed();

Effects: With b as the base engine, invokes b.seed().

void seed(result_type s);

Effects: With b as the base engine, invokes b.seed(s).

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:

(10.1) The complexity of each function shall not exceed the complexity of the corresponding function applied to the base engine.

(10.2) 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.

(10.3) Copying A’s state (e.g., during copy construction or copy assignment) shall include copying the state of the base engine of A.

(10.4) The textual representation of A shall include the textual representation of its base engine.

26.6.2.6 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 26.6.2.6. In that Table and throughout this subclause,

(3.1) \( T \) is the type named by D’s associated result_type;

(3.2) \( P \) is the type named by D’s associated param_type;

(3.3) \( d \) is a value of D, and x and y are (possibly const) values of D;

(3.4) \( \text{glb} \) and \( \text{lub} \) 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;

(3.5) \( p \) is a (possibly const) value of P;

(3.6) \( g, g1, \) and \( g2 \) are lvalues of a type meeting the requirements of a uniform random bit generator (26.6.2.3);

(3.7) \( \text{os} \) is an lvalue of the type of some class template specialization basic_ostream<charT, traits>; and
(3.8) 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 95: Random number distribution requirements

<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><code>T</code></td>
<td><code>T is an arithmetic type (6.8.1)</code>.</td>
<td>compile-time</td>
</tr>
<tr>
<td><code>D::param_type</code></td>
<td><code>P</code></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()</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(p)</code></td>
<td></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><code>P</code></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><code>void</code></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><code>T</code></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><code>T</code></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><code>T</code></td>
<td>Returns <code>glb</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x.max()</code></td>
<td><code>T</code></td>
<td>Returns <code>lub</code>.</td>
<td>constant</td>
</tr>
<tr>
<td><code>x == y</code></td>
<td><code>bool</code></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><code>bool</code></td>
<td><code>(x == y)</code>.</td>
<td>same as <code>x == y</code>.</td>
</tr>
</tbody>
</table>
© ISO/IEC 

Expression | Return type | Pre/post-condition | Complexity |
--- | --- | --- | --- |
\( \text{os} \ll x \) | reference to the type of \( \text{os} \) | Writes to \( \text{os} \) a textual representation for the parameters and the additional internal data of \( x \). Postconditions: The \( \text{os}.\text{fmtflags} \) and fill character are unchanged. | |

\( \text{is} \gg d \) | reference to the type of \( \text{is} \) | Restores from \( \text{is} \) the parameters and additional internal data of the lvalue \( d \). If bad input is encountered, ensures that \( d \) is unchanged by the operation and calls \( \text{is.setstate}(\text{ios\_base::failbit}) \) (which may throw \( \text{ios\_base::failure} \) (29.5.5.4)). Preconditions: \( \text{is} \) provides a textual representation that was previously written using an \( \text{os} \) whose imbued locale and whose type’s template specialization arguments \( \text{charT} \) and \( \text{traits} \) were the same as those of \( \text{is} \). Postconditions: The \( \text{is}.\text{fmtflags} \) are unchanged. | |

4 \( D \) shall meet the \textit{Cpp17CopyConstructible} (Table 29) and \textit{Cpp17CopyAssignable} (Table 31) requirements.

5 The sequence of numbers produced by repeated invocations of \( Cpp17CopyAssign \) (Table 31) shall be independent of any invocation of \( \text{os} \ll d \) or of any \textit{const} member function of \( D \) between any of the invocations \( d(g) \).

6 If a textual representation is written using \( \text{os} \ll x \) and that representation is restored into the same or a different object \( y \) of the same type using \( \text{is} \gg y \), repeated invocations of \( y(g) \) shall produce the same sequence of numbers as would repeated invocations of \( x(g) \).

7 It is unspecified whether \( D::\text{param\_type} \) is declared as a (nested) \texttt{class} or via a \texttt{typedef}. In this subclause \ref{rand.eng}, declarations of \( D::\text{param\_type} \) are in the form of \texttt{typedef}s for convenience of exposition only.

8 \( P \) shall meet the \textit{Cpp17CopyConstructible} (Table 29), \textit{Cpp17CopyAssignable} (Table 31), and \textit{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

\[
\text{using distribution\_type} = D;
\]

\section*{26.6.3 Random number engine class templates} \hfill \textit{rand.eng}

Each type instantiated from a class template specified in this subclause 26.6.3 meets the requirements of a random number engine (26.6.2.4) type.

1 Except where specified otherwise, the complexity of each function specified in this subclause 26.6.3 is constant.

2 Except where specified otherwise, no function described in this subclause 26.6.3 throws an exception.

3 Every function described in this subclause 26.6.3 that has a function parameter \( q \) of type \texttt{Sseq} for a template type parameter named \texttt{Sseq} that is different from type \texttt{seed\_seq} throws what and when the invocation of \( q.g\)enerate throws.

4 Descriptions are provided in this subclause 26.6.3 only for engine operations that are not described in 26.6.2.4 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.

6 Each template specified in this subclause 26.6.3 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.

7 For every random number engine and for every random number engine adaptor X defined in this subclause (26.6.3) and in subclause 26.6.4:

(7.1) — if the constructor

\[
\text{template<class Sseq> explicit X(Sseq& q);}
\]

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

\[
\text{template<class Sseq> void seed(Sseq& q);}
\]

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.

26.6.3.1 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 \(GA(x_i) = x_i + 1\).

```
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);
};
```

2 If the template parameter \(m\) is 0, the modulus \(m\) used throughout this subclause 26.6.3.1 is \(\text{numeric_limits<result_type>::max()} + 1\). [Note: \(m\) need not be representable as a value of type result_type. — end note]

3 If the template parameter \(m\) is not 0, the following relations shall hold: \(a < m\) and \(c < m\).

4 The textual representation consists of the value of \(x_i\).
Let effects: If c mod m is 0 and s mod m is 0, sets the engine’s state to 1, otherwise sets the engine’s state to s mod m.

Effects: With k = ⌈\log_2 m \rceil 32 and a an array (or equivalent) of length k + 3, invokes q.generate(a + 0, a + k + 3) and then computes S = (\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{2j}) mod m. If c mod m is 0 and S is 0, sets the engine’s state to 1, else sets the engine’s state to S.

26.6.3.2 Class template mersenne_twister_engine

A mersenne_twister_engine random number engine produces unsigned integer random numbers in the closed interval [0, 2^m − 1]. The state x_i 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 r, 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, t, c, and ℓ.

The state transition is performed as follows:

(2.1) Concatenate the upper w − r bits of X_{i−n} with the lower r bits of X_{i+1−n} to obtain an unsigned integer value Y.

(2.2) With \alpha = a \cdot (Y \text{ bitand } 1), set X_i to X_{i+m−n} \text{ xor } (Y \text{ rshift } 1) \text{ xor } \alpha.

The sequence X is initialized with the help of an initialization multiplier f.

The generation algorithm determines the unsigned integer values z_1, z_2, z_3, z_4 as follows, then delivers z_4 as its result:

(3.1) Let z_1 = X_i \text{ xor } ((X_i \text{ rshift } a) \text{ bitand } d).

(3.2) Let z_2 = z_1 \text{ xor } ((z_1 \text{ lshift } s) \text{ bitand } b).

(3.3) Let z_3 = z_2 \text{ xor } ((z_2 \text{ lshift } t) \text{ bitand } c).

(3.4) Let z_4 = z_3 \text{ xor } (z_3 \text{ rshift } ℓ).

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 {

public:

using result_type = UIntType;

// engine characteristics
static constexpr size_t word_size = w;
static constexpr size_t state_size = n;
static constexpr size_t shift_size = m;
static constexpr size_t mask_bits = r;
static constexpr UIntType xor_mask = a;
static constexpr size_t tempering_u = u;
static constexpr UIntType tempering_d = d;
static constexpr size_t tempering_s = s;
static constexpr UIntType tempering_b = b;
static constexpr size_t tempering_t = t;
static constexpr UIntType tempering_c = c;
static constexpr size_t tempering_l = l;
static constexpr UIntType initialization_multiplier = f;
static constexpr result_type min() { return 0; }
static constexpr result_type max() { return 2^w − 1; }

243) 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.


static constexpr result_type default_seed = 5489u;

// constructors and seeding functions

mersenne_twister_engine() : mersenne_twister_engine(default_seed) {}
explicit mersenne_twister_engine(result_type value);
template<class Sseq> explicit mersenne_twister_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: 0 < n, m <= n, 2u < w, r <= w, u <= w, s <= w, t <= w, l <= w, u <= numeric_limits<UIntType>::digits, a <= (1u<<w) - 1u, b <= (1u<<w) - 1u, c <= (1u<<w) - 1u, d <= (1u<<w) - 1u, and f <= (1u<<w) - 1u.

The textual representation of \(x_i\) consists of the values of \(X_{i-n}, \ldots, X_{i-1}\), in that order.

explicit mersenne_twister_engine(result_type value);

Effects: Sets \(X_{-n}\) to value mod \(2^w\). Then, iteratively for \(i = 1-n, \ldots, -1\), sets \(X_i\) to

\[
\left( f \cdot (X_{i-1} \text{xor} (X_{i-1} \text{rshift} (w-2))) + i \mod n \right) \mod 2^w.
\]

Complexity: \(\Theta(n)\).

template<class Sseq> explicit mersenne_twister_engine(Sseq& q);

Effects: With \(k = \lceil w/32 \rceil\) and \(a\) an array (or equivalent) of length \(n \cdot k\), invokes \(q\).\text{generate}(a + 0, a + n \cdot k)\) and then, iteratively for \(i = -n, \ldots, -1\), sets \(X_i\) to \(\left( \sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j} \right) \mod 2^w\). Finally, if the most significant \(w - r\) bits of \(X_{-n}\) are zero, and if each of the other resulting \(X_i\) is 0, changes \(X_{-n}\) to \(2^{w-1}\).

26.6.3.3 Class template subtract_with_carry_engine

A subtract_with_carry_engine random number engine produces unsigned integer random numbers.

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:

\begin{align}
\text{(1.1)} & \quad \text{Let } Y = X_{i-s} - X_{i-r} - c. \\
\text{(1.2)} & \quad \text{Set } X_i \text{ to } y = Y \mod m. \text{ Set } c \text{ to 1 if } Y < 0, \text{ otherwise set } c \text{ to 0.}
\end{align}

Note: 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 \(m^r - m^s + 1\) and \(a = b - (b - 1)/m\). — end note

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.

template<class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine {
public:
  // types
  using result_type = UIntType;

  // engine characteristics
  static constexpr size_t word_size = w;
  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 = 19780503u;

§ 26.6.3.3 1165
// 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:

\[
\text{linear_congruential_engine<result_type, } 40014u, 0u, 2147483563u > e(\text{value == 0u } ? \text{default_seed} : \text{value})
\]

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.

26.6.4 Random number engine adaptor class templates

26.6.4.1 In general

Each type instantiated from a class template specified in this subclause 26.6.4 meets the requirements of a random number engine adaptor (26.6.2.5) type.
Except where specified otherwise, the complexity of each function specified in this subclause 26.6.4 is constant.
Except where specified otherwise, no function described in this subclause 26.6.4 throws an exception.
Every function described in this subclause 26.6.4 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.

Descriptions are provided in this subclause 26.6.4 only for adaptor operations that are not described in subclause 26.6.2.5 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.
Each template specified in this subclause 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.

26.6.4.2 Class template discard_block_engine

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.
2 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 \( e_{j+1} \).

3 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);

private:
  Engine e;  // exposition only
  int n;     // exposition only
};
```

4 The following relations shall hold: \( 0 < r \) and \( r \leq p \).

5 The textual representation consists of the textual representation of \( e \) followed by the value of \( n \).

6 In addition to its behavior pursuant to subclause 26.6.2.5, each constructor that is not a copy constructor sets \( n \) to 0.

26.6.4.3 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:

1. Let \( R = e \cdot \text{max()} - e \cdot \text{min()} + 1 \) and \( m = \lceil \log_2 R \rceil \).
2. With \( n \) as determined below, let \( w_0 = \lceil w/n \rceil \), \( n_0 = n - w \mod n \), \( y_0 = 2^{w_0} \lceil R/2^{w_0} \rceil \), and \( y_1 = 2^{w_0+1} \lceil R/2^{w_0+1} \rceil \).
3. Let \( n = \lceil w/m \rceil \) if and only if the relation \( R - y_0 \leq \lceil y_0/n \rceil \) holds as a result. Otherwise let \( n = 1 + \lceil w/m \rceil \).

[Note: 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 \( y_0 + e \cdot \text{min()} \) and \( n - n_0 \) values less than \( y_1 + e \cdot \text{min()} \).
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 \( (k = 0; k \neq n_0; k += 1) \) {
\[
\text{do } u = e() - e.min(); \text{ while } (u \geq y_0);
S = 2^{w_0} \cdot S + u \mod 2^{w_0};
\]
}

for \( (k = n_0; k \neq n; k += 1) \) {
\[
\text{do } u = e() - e.min(); \text{ while } (u \geq y_1);
S = 2^{w_0+1} \cdot S + u \mod 2^{w_0+1};
\]
}

```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 \( e \).

### 26.6.4.4 Class template shuffle_order_engine

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 \( x \) of a `shuffle_order_engine` engine adaptor object \( x \) consists of the state \( e \), 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 \( j = \left\lfloor \frac{k \cdot (Y - e.min)}{e.max - e.min + 1} \right\rfloor \).
2. Set \( Y \) to \( V_j \) and then set \( V_j \) to \( e() \).
3. 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);
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 26.6.2.5, each constructor that is not a copy constructor initializes V[0],...,V[k-1] and Y, in that order, with values returned by successive invocations of e().

26.6.5 Engines and engine adaptors with predefined parameters [rand.predef]

using minstd_rand0 =
    linear_congruential_engine<uint_fast32_t, 16'807, 0, 2'147'483'647>;
1 Required behavior: The 10000th 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 10000th 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, 0xdef6'0000, 18, 1'812'433'253>;
3 Required behavior: The 10000th 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, 0x5s55'5s5s's5ss's5ss, 17,
    0x71d6'7fff'eda8'0000, 37, 0xffffffff, 43, 6'364'136'223'846'793'005>;
4 Required behavior: The 10000th 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 ranlux24_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 ranlux48_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: The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this typedef need not generate identical sequences across implementations. — end note]

26.6.6 Class random_device

A random_device uniform random bit generator produces nondeterministic random numbers.

If implementation limitations prevent generating nondeterministic random numbers, the implementation 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;
};
explicit random_device(const string& token);

Remarks: The semantics of the token parameter and the token value used by the default constructor are implementation-defined.\(^{244}\)

Throws: A value of an implementation-defined type derived from exception if the random_device could not be initialized.

double entropy() const noexcept;

Remarks: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate\(^{245}\) for the random numbers returned by operator(), in the range \(\min()\) to \(\log(\max() + 1)\).

result_type operator()();

Returns: A nondeterministic random value, uniformly distributed between \(\min()\) and \(\max()\) (inclusive). It is implementation-defined how these values are generated.

Throws: A value of an implementation-defined type derived from exception if a random number could not be obtained.

26.6.7 Utilities [rand.util]

26.6.7.1 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
};

Postconditions: v.empty() is true.

Throws: Nothing.

template<class T>
seed_seq(initializer_list<T> il);

Mandates: T is an integer type.

Effects: Same as seed_seq(il.begin(), il.end()).

\(^{244}\) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\(^{245}\) 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 (23.3.5.2).

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 (23.3.5.6) 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 rshift 27):

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 ≥ 623) ? 11 : (n ≥ 68) ? 7 : (n ≥ 39) ? 5 : (n ≥ 7) ? 3 : (n − 1)/2;

With m as the larger of s + 1 and n, transform the elements of the range: iteratively for k = 0, . . . , m − 1, calculate values

r1 = 1664525 · T(begin[k] xor begin[k + p] xor begin[k − 1])

r2 = r1 + \begin{cases} s \\ k \mod n + v[k − 1] \\ k \mod n \end{cases} , \begin{cases} 0 < k ≤ s \\ s < k \end{cases}

and, in order, increment begin[k + p] by r1, increment begin[k + q] by r2, and set begin[k] to r2.

Transform the elements of the range again, beginning where the previous step ended: iteratively for k = m, . . . , m + n − 1, calculate values

r3 = 1566083941 · T(begin[k] + begin[k + p] + begin[k − 1])

r4 = r3 − (k mod n)

and, in order, update begin[k + p] by xoring it with r3, update begin[k + q] by xoring it with r4, and set begin[k] to r4.

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 (23.3.5.3).

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.
26.6.7.2 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 \( b \) is the lesser of \texttt{numeric_limits<RealType>::digits} and \( \texttt{bits} \), and \( R \) is the value of \( \texttt{g.max()} - \texttt{g.min()} + 1 \).

2 Effects: Invokes \( g() \) \( k \) times to obtain values \( g_0, \ldots, g_{k-1} \), respectively. Calculates a quantity

\[
S = \sum_{i=0}^{k-1} (g_i - \texttt{g.min()}) \cdot R^i
\]

using arithmetic of type \( \texttt{RealType} \).

3 Returns: \( S/R^k \). \[Note: 0 \leq S/R^k < 1. —end note\]

4 Throws: What and when \( g \) throws.

5 \[Note: 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\]

26.6.8 Random number distribution class templates

26.6.8.1 In general

1 Each type instantiated from a class template specified in this subclause 26.6.8 meets the requirements of a random number distribution (26.6.2.6) type.

2 Descriptions are provided in this subclause 26.6.8 only for distribution operations that are not described in 26.6.2.6 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.

26.6.8.2 Uniform distributions

26.6.8.2.1 Class template uniform_int_distribution

1 A \texttt{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<class URBG>
result_type operator()(URBG& g);
template<class URBG>
result_type operator()(URBG& g, const param_type& parm);
```

\( b \) is introduced to avoid any attempt to produce more bits of randomness than can be held in \( \texttt{RealType} \).
// 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.

26.6.8.2.2 Class template uniform_real_distribution [rand.dist.uni.real]
A uniform_real_distribution random number distribution produces random numbers x, a ≤ x < b, distributed according to the constant probability density function
\[ p(x | a, b) = \frac{1}{b - a} \, . \]

[Note: This implies that p(x | 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 ≤ 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.
result_type b() const;

Returns: The value of the b parameter with which the object was constructed.

26.6.8.3 Bernoulli distributions

26.6.8.3.1 Class bernoulli_distribution

A bernoulli_distribution random number distribution produces bool values b distributed according to the discrete probability function

\[ P(b | p) = \begin{cases} p & \text{if } b = \text{true}, \\ 1 - p & \text{if } b = \text{false}. \end{cases} \]

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;
  }

explicit bernoulli_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.

26.6.8.3.2 Class template binomial_distribution

A binomial_distribution random number distribution produces integer values i ≥ 0 distributed according to the discrete probability function

\[ P(i | t, p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{t-i}. \]

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.

26.6.8.3.3 Class template geometric_distribution
A geometric_distribution random number distribution produces integer values i ≥ 0 distributed according
to the discrete probability function

\[ P(i \mid p) = p \cdot (1 - p)^i \]

1

template<class IntType = int>
class geometric_distribution {
public:
    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);

2 Preconditions: $0 < p < 1$.
3 Remarks: $p$ corresponds to the parameter of the distribution.

double p() const;
4 Returns: The value of the $p$ parameter with which the object was constructed.

26.6.8.3.4 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

$$P(i|k,p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i.$$ 

[Note: This implies that $P(i|k,p)$ 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);

2 Preconditions: $0 < p \leq 1$ and $0 < k$.
3 Remarks: $k$ and $p$ correspond to the respective parameters of the distribution.

IntType k() const;
4 Returns: The value of the $k$ parameter with which the object was constructed.

double p() const;
5 Returns: The value of the $p$ parameter with which the object was constructed.

26.6.8.4 Poisson distributions

26.6.8.4.1 Class template poisson_distribution

A poisson_distribution random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i|\mu) = \frac{e^{-\mu} \mu^i}{i!}.$$ 

The distribution parameter $\mu$ is also known as this distribution’s mean.
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);

Preconditions: 0 < mean.
Remarks: mean corresponds to the parameter of the distribution.

double mean() const;
Returns: The value of the mean parameter with which the object was constructed.

26.6.8.4.2 Class template exponential_distribution [rand.dist.poiexp]

An exponential_distribution random number distribution produces random numbers \(x > 0\) distributed according to the probability density function
\[
p(x \mid \lambda) = \lambda e^{-\lambda x}.
\]

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);

        // property functions
        RealType lambda() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
);

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.

26.6.8.4.3 Class template gamma_distribution

A gamma_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | \alpha, \beta) = \frac{e^{-x/\beta} \cdot \beta^\alpha \cdot \Gamma(\alpha) \cdot x^{\alpha-1}}{\beta^\alpha \cdot \Gamma(\alpha)}.
\]

template<class RealType = double>
class gamma_distribution {
public:

// types
using result_type = RealType;
using param_type = unspecified;

// constructors and reset functions
gamma_distribution() : gamma_distribution(1.0) {}
explicit gamma_distribution(RealType alpha, RealType beta = 1.0);
explicit gamma_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 alpha() const;
RealType beta() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

};

explicit gamma_distribution(RealType alpha, RealType beta = 1.0);

Preconditions: 0 < alpha and 0 < beta.

Remarks: alpha and beta correspond to the parameters of the distribution.

RealType alpha() const;

Returns: The value of the alpha parameter with which the object was constructed.

RealType beta() const;

Returns: The value of the beta parameter with which the object was constructed.
26.6.8.4.4 Class template `weibull_distribution`

A `weibull_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x | a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^a\right).$$

```cpp
template<class RealType = double>
class weibull_distribution {
    public:
        // 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 $0 < b$.
3  Remarks: $a$ and $b$ correspond to the respective parameters of the distribution.

```cpp
template<class RealType = double>
class extreme_value_distribution {
    public:
        // types
        using result_type = RealType;
        using param_type = unspecified;

        // constructor and reset functions
        extreme_value_distribution(); extreme_value_distribution(1.0); explicit extreme_value_distribution(RealType a, RealType b = 1.0);
        explicit extreme_value_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;
};
```

4  Returns: The value of the `a` parameter with which the object was constructed.
5  Returns: The value of the `b` parameter with which the object was constructed.

26.6.8.4.5 Class template `extreme_value_distribution`

An `extreme_value_distribution` random number distribution produces random numbers $x$ distributed according to the probability density function

$$p(x | a, b) = \frac{1}{b} \cdot \exp\left(\frac{a-x}{b} - \exp\left(\frac{a-x}{b}\right)\right).$$

```cpp
template<class RealType = double>
class extreme_value_distribution {
    public:
        // types
        using result_type = RealType;
        using param_type = unspecified;

        template<class RealType = double>
        class extreme_value_distribution {
        public:
            // types
            using result_type = RealType;
            using param_type = unspecified;

247) 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.
// constructor and reset functions
extreme_value_distribution() : extreme_value_distribution(0.0) {}
explicit extreme_value_distribution(RealType a, RealType b = 1.0);
explicit extreme_value_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 extreme_value_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.

26.6.8.5 Normal distributions
26.6.8.5.1 Class template normal_distribution

A normal_distribution random number distribution produces random numbers x distributed according to
the probability density function
\[
\frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right)
\]
The distribution parameters \( \mu \) and \( \sigma \) are also known as this distribution’s mean and standard deviation.

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;

§ 26.6.8.5.1
template<class RealType = double>
class lognormal_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;
    // constructor and reset functions
    explicit 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);

2 Preconditions: 0 < s.

3 Remarks: m and s correspond to the respective parameters of the distribution.

RealType m() const;
4 Returns: The value of the m parameter with which the object was constructed.

RealType s() const;
5 Returns: The value of the s parameter with which the object was constructed.
26.6.8.5.3 Class template chi_squared_distribution

A `chi_squared_distribution` random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}.
\]

```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) {} // 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;
    };
```

2

Preconditions: \( 0 < n \).

Remarks: \( n \) corresponds to the parameter of the distribution.

RealType n() const;

Returns: The value of the \( n \) parameter with which the object was constructed.

26.6.8.5.4 Class template cauchy_distribution

A `cauchy_distribution` random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x | a, b) = \left( \pi b \left( 1 + \left( \frac{x-a}{b} \right)^2 \right) \right)^{-1}.
\]

```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) {} // 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.

26.6.8.5.5 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

\[
p(x | m, n) = \frac{\Gamma\left(\frac{m+n}{2}\right)}{\Gamma\left(\frac{m}{2}\right) \Gamma\left(\frac{n}{2}\right)} \cdot \left(\frac{m}{n}\right)^{m/2} \cdot x^{(m/2)-1} \cdot \left(1 + \frac{mx}{n}\right)^{-(m+n)/2}.
\]

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) {};
    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);

    Preconditions: 0 < m and 0 < n.
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.

### 26.6.8.5.6 Class template student_t_distribution

A `student_t_distribution` random number distribution produces random numbers \(x\) distributed according to the probability density function

\[
p(x | n) = \frac{1}{\sqrt{n\pi}} \frac{\Gamma((n + 1)/2)}{\Gamma(n/2)} \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}.
\]

```cpp
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) {} // default constructor
    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;
};
```

#### Preconditions: \(n < 0\).

#### Remarks: \(n\) corresponds to the parameter of the distribution.

RealType \(n()\) const;

Returns: The value of the \(n\) parameter with which the object was constructed.

### 26.6.8.6 Sampling distributions

#### 26.6.8.6.1 Class template discrete_distribution

A `discrete_distribution` random number distribution produces random integers \(i, 0 \leq i < n\), distributed according to the discrete probability function

\[
P(i | p_0, \ldots, p_{n-1}) = p_i.
\]

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 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 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();

Effects: Constructs a discrete_distribution object with n = 1 and p0 = 1. [Note: Such an object will always deliver the value 0. — end note]

discrete_distribution(InputIterator firstW, InputIterator lastW);

Mandates: is_convertible_v<iterator_traits<InputIterator>::value_type, double> is true.

Preconditions: InputIterator meets the Cpp17InputIterator requirements (23.3.5.2). If firstW == lastW, let n = 1 and w0 = 1. Otherwise, (firstW, lastW) forms a sequence w of length n > 0.

Effects: Constructs a discrete_distribution object with probabilities given by the formula above.

discrete_distribution(initializer_list<double> wl);

Effects: Same as discrete_distribution(wl.begin(), wl.end()).

discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);

Mandates: is_invocable_r_v<double, UnaryOperation&, double> is true.

Preconditions: If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax - xmin)/n holds.

Effects: Constructs a discrete_distribution object with probabilities given by the formula above, using the following values: If nw = 0, let w0 = 1. Otherwise, let wk = fw(xmin + k·δ + δ/2) for k = 0, ..., n - 1.

Complexity: The number of invocations of fw does not exceed n.

vector<double> probabilities() const;

Returns: A vector<double> whose size member returns n and whose operator[] member returns pk when invoked with argument k for k = 0, ..., n - 1.
26.6.8.6.2 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

\[
p(x | b_0, ..., b_n, \rho_0, ..., \rho_{n-1}) = \rho_i, \text{ for } b_i \leq x < b_{i+1}.
\]

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 \( i = 0, ..., n - 1 \). Unless specified otherwise, the remaining \( n \) distribution parameters are calculated as:

\[
\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, ..., n - 1,
\]

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: \( 0 < S = w_0 + \cdots + w_{n-1} \).

```cpp
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 \( b_1 = 1 \).

```cpp
template<class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                InputIteratorW firstW);
```

**Mandates:** Both of

\[(4.1)\]  
- `is_convertible_v<iterator_traits<InputIteratorB>::value_type, double>`  
\[(4.2)\]  
- `is_convertible_v<iterator_traits<InputIteratorW>::value_type, double>`
are true.

Preconditions: InputIteratorB and InputIteratorW each meet the Cpp17InputIterator requirements (23.3.5.2). If firstB == lastB or ++firstB == lastB, let n = 1, w₀ = 1, b₀ = 0, and b₁ = 1. Otherwise, [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 wᵦ for k ≥ 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 bl.size() < 2, let n = 1, w₀ = 1, b₀ = 0, and b₁ = 1. Otherwise, let [bl.begin(), bl.end()] form a sequence b₀,...,bₙ, and let wᵦ = fw((bᵦ₊₁ + bᵦ) / 2) for k = 0,...,n−1.

Complexity: The number of invocations of 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 nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax − xmin)/n holds.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: Let b_k = xmin + kδ for k = 0,...,n, and w_k = fw(b_k + δ/2) for k = 0,...,n−1.

Complexity: The number of invocations of 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 ρ_k when invoked with argument k for k = 0,...,n − 1.

26.6.8.6.3 Class template piecewise_linear_distribution

A piecewise_linear_distribution random number distribution produces random numbers x, b₀ ≤ x < bₙ, distributed over each subinterval [bᵦ, bᵦ₊₁) according to the probability density function

\[
p(x | b₀,...,bₙ, ρ₀,...,ρₙ) = \rhoᵦ \frac{bᵦ₊₁ - x}{bᵦ₊₁ - bᵦ} + \rhoᵦ₊₁ \frac{x - bᵦ}{bᵦ₊₁ - bᵦ}, \text{ for } bᵦ ≤ x < bᵦ₊₁.\]

The n + 1 distribution parameters bᵦ, also known as this distribution’s internal boundaries, shall satisfy the relation bᵦ < bᵦ₊₁ for i = 0,...,n − 1. Unless specified otherwise, the remaining n + 1 distribution parameters are calculated as ρᵦ = wᵦ / S for k = 0,...,n, in which the values wᵦ, commonly known as the weights at boundaries, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

\[
0 < S = \frac{1}{2} \cdot \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k).
\]

```
template<class RealType = double>
class piecewise_linear_distribution {
public:
  // types
  using result_type = RealType;
  using param_type = unspecified;

  // constructor and reset functions
  template<class InputIteratorB, class InputIteratorW>
  piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                      InputIteratorW firstW);

  template<class UnaryOperation>
  piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);

  // constructor and reset functions
  piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

  // member functions
  vector<result_type> densities() const;
};
```
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: \( \text{is\_invocable\_r\_v\_v\_double, \text{UnaryOperation}\_\_\_double} \) is true.
Preconditions: \( \text{InputIteratorB and InputIteratorW} \) each meet the Cpp17InputIterator requirements (23.3.5.2). If \( \text{firstB == lastB or ++firstB == lastB, let n = 1, \rho_0 = \rho_1 = 1, b_0 = 0, \) and \( b_1 = 1. \) Otherwise, \([\text{firstB, lastB}]\) forms a sequence \( b \) of length \( n + 1 \), the length of the sequence \( w \) starting from \( \text{firstW} \) 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: \( \text{is\_invocable\_r\_v\_double, \text{UnaryOperation}\_\_\_double} \) 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 \( b_0, ..., b_n \), and let \( w_k = \text{fw}(b_k) \) for \( k = 0, ..., n. \)
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: \( \text{is\_invocable\_r\_v\_double, \text{UnaryOperation}\_\_\_double} \) 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 \( k = 0, ..., n, \) and \( w_k = \text{fw}(b_k) \) for \( k = 0, ..., n. \)
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 \( k = 0, ..., n. \)
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 \( k = 0, \ldots, n \).

26.6.9 Low-quality random number generation

[Note: 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 (16.5.5.10). [Note: 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;
  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 indirection 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>&);
  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

1190
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>&);

§ 26.7.1
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<bool> operator>=(const typename valarray<T>::value_type&, const valarray<T>&);

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.

Implementations introducing such replacement types shall provide additional functions and operators as follows:

1. 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;
2. for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

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` (17.6.3.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.
valarray& operator-= (const T&);
valarray& operator^= (const T&);
valarray& operator&= (const T&);
valarray& operator|= (const T&);
valarray& operator<<=(const T&);
valarray& operator>>=(const T&);
valarray& operator*= (const valarray&);
valarray& operator/= (const valarray&);
valarray& operator%= (const valarray&);
valarray& operator+= (const valarray&);
valarray& operator-= (const valarray&);
valarray& operator^= (const valarray&);
valarray& operator|= (const valarray&);
valarray& operator&= (const valarray&);
valarray& operator<<=(const valarray&);
valarray& operator>>=(const valarray&);

// 26.7.2.8, member functions
void swap(valarray&) noexcept;

size_t size() const;
T sum() const;
T min() const;
T max() const;

valarray shift (int) const;
valarray cshift(int) const;
valarray apply(T func(T)) const;
valarray apply(T func(const T&)) const;
void resize(size_t sz, T c = T());
};

template<class T, size_t cnt> valarray(const T(&)[cnt], size_t) -> valarray<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.\(^{249}\)

### 26.7.2.2 Constructors

[valarray.cons]

valarray();

[249] 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.

250) This default constructor is essential, since arrays of valarray may be useful. After initialization, the length of an empty array can be increased with the resize member function.

---

valarray(const T& v, size_t n);

Effects: Constructs a valarray that has length n. Each element of the array is value-initialized (9.4).

valarray(const T* p, size_t n);

Effects: Constructs a valarray that has length n. Each element of the array is initialized with v.

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.\(^{251}\)

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 \( v \).\(^ {252}\)

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.assign]

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.

\(^{251}\) This constructor is the preferred method for converting a C array to a valarray object.

\(^{252}\) 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.
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);

Preconditions: n < size() is true.

Returns: A reference to the corresponding element of the array. [Note: The expression (a[i] = q, a[i]) == q evaluates to true for any non-constant valarray<T> a, any T q, and for any size_t i such that the value of i is less than the length of a. — end note]

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 < 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 < a.size() and j < b.size(). [Note: This property indicates an absence of aliasing and may be used to advantage by optimizing compilers. Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from operator new, and other techniques to generate efficient valarrays. — 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

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;

Returns: A valarray containing those elements of the controlled sequence designated by slicearr.
[Example:
const valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
— end example]

slice_array<T> operator[](slice slicearr);

Returns: An object that holds references to elements of the controlled sequence selected by slicearr.
[Example:
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdEBgCjkDmnEp", 16);
— end example]

valarray operator[](const gslice& gslicearr) const;

Returns: A valarray containing those elements of the controlled sequence designated by gslicearr.
[Example:
const valarray<char> v0("abcdefghijklmnop", 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>("dfholmo", 6)
— end example]
galice_array<T> operator[] (const gslice& gslicearr);

Returns: An object that holds references to elements of the controlled sequence selected by gslicearr.

[Example:
    valarray<char> v0("abcdefgijklmnop", 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:
    const valarray<char> v0("abcdefgijklmnop", 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:
    valarray<char> v0("abcdefgijklmnop", 16);
    valarray<char> v1("ABCDE", 5);
    const size_t vi[] = { 7, 5, 2, 3, 8 };
    // v0[valarray<size_t>(vi, 5)] returns
    // valarray<char>("hfcdi", 5)
—end example]

valarray operator[](const valarray<size_t>& indarr) const;

Returns: A valarray containing those elements of the controlled sequence designated by indarr.

[Example:
    const valarray<char> v0("abcdefgijklmnop", 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:
    valarray<char> v0("abcdefgijklmnop", 16);
    valarray<char> v1("ABCDE", 5);
    const size_t vi[] = { 7, 5, 2, 3, 8 };
    v0[valarray<size_t>(vi, 5)] = v1;
    // v0 == valarray<char>("abCDeBgAEijklmnop", 16)
—end example]

26.7.2.6 Unary operators

valarray operator+() const;
valarray operator-() const;
valarray operator~() const;

[valarray.unary]
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.cassign]

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);
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);
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 [valarray.members]

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: 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: 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

```cpp
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 lengths 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&);
```

```
operator% (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
operator+ (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
operator- (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
operator^ (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
operator& (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
operator| (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
operator<< (const valarray<T>&, const typename valarray<T>::value_type&);
```

```
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.

### Logical operators

```
template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
```

```
operator!=(const valarray<T>&, const valarray<T>&);
```

```
operator< (const valarray<T>&, const valarray<T>&);
```

```
operator> (const valarray<T>&, const valarray<T>&);
```

```
operator<=(const valarray<T>&, const valarray<T>&);
```

```
operator>=(const valarray<T>&, const valarray<T>&);
```

```
operator&&(const valarray<T>&, const 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 `bool` or which can be unambiguously implicitly converted to `bool`.

§ 26.7.3.2
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.

```cpp
template<class T> valarray<bool> operator==(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator==(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator!=(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator!=(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator<(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator<(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator<=(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator>(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator>(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator>=(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator>=(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator&&(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator&&(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<bool> operator||(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<bool> operator||(const typename valarray<T>::value_type&, 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.

Returns: A `valarray<bool>` 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 and the non-array argument.

26.7.3.3 Transcendentals

```cpp
template<class T> valarray<T> abs (const valarray<T>&);
```

```cpp
template<class T> valarray<T> acos (const valarray<T>&);
```

```cpp
template<class T> valarray<T> asin (const valarray<T>&);
```

```cpp
template<class T> valarray<T> atan (const valarray<T>&);
```

```cpp
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
```

```cpp
template<class T> valarray<T> atan2(const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<T> atan2(const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<T> cos (const valarray<T>&);
```

```cpp
template<class T> valarray<T> cosh (const valarray<T>&);
```

```cpp
template<class T> valarray<T> exp (const valarray<T>&);
```

```cpp
template<class T> valarray<T> log (const valarray<T>&);
```

```cpp
template<class T> valarray<T> log10(const valarray<T>&);
```

```cpp
template<class T> valarray<T> pow (const valarray<T>&, const valarray<T>&);
```

```cpp
template<class T> valarray<T> pow (const valarray<T>&, const typename valarray<T>::value_type&);
```

```cpp
template<class T> valarray<T> pow (const typename valarray<T>::value_type&, const valarray<T>&);
```

```cpp
template<class T> valarray<T> sin (const valarray<T>&);
```

```cpp
template<class T> valarray<T> sinh (const valarray<T>&);
```

```cpp
template<class T> valarray<T> sqrt (const valarray<T>&);
```

§ 26.7.3.3

[valarray.transcend]
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

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

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.\footnote{BLAS stands for Basic Linear Algebra Subprograms. C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammerling: 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.}

26.7.4.2 Constructors

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: 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

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

friend bool operator==(const slice& x, const slice& y);

Effects: Equivalent to:

return x.start() == y.start() && x.size() == y.size() && x.stride() == y.stride();
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: 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.
26.7.6 The gslice class

[© ISO/IEC N4861 26.7.6 The gslice class]
26.7.6.1 Overview

```cpp
namespace std {
    class gslice {
        public:
            gslice();
            gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);
            size_t start() const;
            valarray<size_t> size() const;
            valarray<size_t> stride() const;
    };
}
```

1 This class represents a generalized slice out of an array. A `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.

2 A `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 `valarray` template, which is one-dimensional. The set of one-dimensional index values specified by a `gslice` are

\[
k = s + \sum_j i_j d_j
\]

where the multidimensional indices \( i_j \) range in value from 0 to \( l_j - 1 \).

3 [Example: The gslice specification]

```cpp
start = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

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:

\[
(i_0, \ i_1, \ i_2, \ 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)
```

That is, the highest-ordered index turns fastest. — end example]

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 [Example: 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

\[
(0, \ 0, \ 0, \ 3),
(0, \ 0, \ 1, \ 4),
... 
```
If a degenerate slice is used as the argument to the non-\texttt{const} version of \texttt{operator[]} (\texttt{const gslice&}), the behavior is undefined.

\subsection*{26.7.6.2 Constructors} [gslice.cons]

\begin{verbatim}
gslice();
gslice(size_t start, const valarray<size_t>& lengths, const valarray<size_t>& strides);
gslice(const gslice&);
\end{verbatim}

1 The default constructor is equivalent to \texttt{gslice(0, valarray<size_t>(), valarray<size_t>())}.

The constructor with arguments builds a \texttt{gslice} based on a specification of start, lengths, and strides, as explained in the previous subclause.

\subsection*{26.7.6.3 Access functions} [gslice.access]

\begin{verbatim}
size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
\end{verbatim}

1 \textit{Returns:} The representation of the start, lengths, or strides specified for the \texttt{gslice}.

2 \textit{Complexity:} \texttt{start()} is constant time. \texttt{size()} and \texttt{stride()} are linear in the number of strides.

\subsection*{26.7.7 Class template \texttt{gslice_array}} [template.gslice.array]

\subsection*{26.7.7.1 Overview} [template.gslice.array.overview]

\begin{verbatim}
namespace std {
  template<class T> class gslice_array {
    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
  
  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
  
};
\end{verbatim}

1 This template is a helper template used by the \texttt{gslice} subscript operator

\begin{verbatim}
gslice_array<T> valarray<T>::operator[](const gslice&);
\end{verbatim}

2 It has reference semantics to a subset of an array specified by a \texttt{gslice} object. Thus, the expression \texttt{a[gslice(1, length, stride)] = b} has the effect of assigning the elements of \texttt{b} to a generalized slice of the elements in \texttt{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;

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 it 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;

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 mask object refers.

26.7.8.4 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 mask_array object refers.

26.7.9 Class template indirect_array

26.7.9.1 Overview

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&); const;
            void operator=(const T&) const;

            indirect_array() = delete; // as implied by declaring copy constructor above
        }
    };

1 This template is a helper template used by the indirect subscript operator
    indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).

2 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;

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 it refers.

2 If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

[Example:

    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;

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 indirect_array object refers.

2 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;

1 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 (23.3.5.6) and models contiguous_iterator (23.3.4.14), whose value_type is the template parameter T and whose reference type is T*. 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 T*.

1 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);

3 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);

4 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 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 expm1f(float x);
long double expm1l(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);   // see 16.2
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);
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);
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);
float log1pf(float x);
long double log1pl(long double x);

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 cbert(float x); // see 16.2
double cbert(double x);
long double cbrt(long double x); // see 16.2
float cbrtf(float x);
long double cbrtl(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 erfl(long double x);

float erfc(float x); // see 16.2
double erfc(double x);
long double erfc(long double x); // see 16.2
float erfcf(float x);
long double erfcf(long double x);

float lgamma(float x); // see 16.2
double lgamma(double x);
long double lgamma(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 tgamma(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);

double floor(double x); // see 16.2
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);
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);
float 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);
float 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);
float 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);
float 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 remainderf(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 remquodouble x, double y, int* quo);
long double remquolong double x, long double y, int* quo);  // see 16.2
float remquofloat x, float y, int* quo);
long double remquolong double x, long double y, int* quo);

float copysign(float x, float y);  // see 16.2
double copysign(double x, double y);
long double copysign(long double x, long double y);  // see 16.2
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

double nan(const char* tagp);
float nanf(const char* tagp);
long double nanl(const char* tagp);

float nextafter(float x, float y);  // see 16.2
double nextafter(double x, double y);
long double nextafter(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 nexttoward(float x, long double y);  // see 16.2
double nexttoward(double x, long double y);
long double nexttoward(long double x, long double y);  // see 16.2
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

float fdim(float x, float y);  // see 16.2
double fdim(double x, double y);
long double fdim(long double x, long double y);  // see 16.2
float fdimf(float x, float y);
long double fdimal(long double x, long double y);

float fmax(float x, float y);  // see 16.2
double fmax(double x, double y);
long double fmalong 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 fmin(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 fmadouble x, double y, double z);
long double fma(long double x, long double y, long double z);  // see 16.2
float fmalong double x, long double y, long double z);

// 26.8.4, linear interpolation
constexpr float lerpfloat a, float b, float t) noexcept;
constexpr double lerndouble a, double b, double t) noexcept;
constexpr long double lerp(long double a, long double b, long double t) noexcept;

§ 26.8.1
// 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 islesseq(float x, float y);
bool islesseq(double x, double y);
bool islesseq(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

// 26.8.6.1, 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);

// 26.8.6.2, 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);

// 26.8.6.3, beta function
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// 26.8.6.4, 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);

// 26.8.6.5, 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);

// 26.8.6.6, 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);

// 26.8.6.7, 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);

// 26.8.6.8, 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);

// 26.8.6.9, 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);

// 26.8.6.10, 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);

// 26.8.6.11, 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);

// 26.8.6.12, 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);

// 26.8.6.13, 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);

// 26.8.6.14, exponential integral
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// 26.8.6.15, Hermite polynomials
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
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: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (16.2). — end note]

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 (6.8.1) 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: Otherwise, all arguments of arithmetic type corresponding to `double` parameters have type `float`. — end note]

[Note: `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: The headers `<cstdlib>` (17.2.2) and `<cmath>` (26.8.1) declare the functions described in this subclause. — end note]

```cpp
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);

2  Effects: The abs functions have the semantics specified in the C standard library for the functions abs, labs, llabs, fabsf, fabs, and fabsl.

3  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 (7.3.6), the program is ill-formed. [Note: 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);

1  Returns: $\sqrt{x^2 + y^2 + z^2}$.

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;

1  Returns: $a + t(b - a)$.

2  Remarks: Let r be the value returned. If isfinite(a) && isfinite(b), then:

(2.1)  — If t == 0, then r == a.
(2.2)  — If t == 1, then r == b.
(2.3)  — If t >= 0 && t <= 1, then isfinite(r).
(2.4)  — If isfinite(t) && a == b, then r == a.
(2.5)  — 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 t1 and t2, 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

1  If any argument value to any of the functions specified in this subclause 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:

(1.1)  — the function description’s Returns: clause explicitly specifies a domain and those argument values fall outside the specified domain, or
(1.2)  — the corresponding mathematical function value has a nonzero imaginary component, or
(1.3)  — the corresponding mathematical function is not mathematically defined.254

2  Unless otherwise specified, each function is defined for all finite values, for negative infinity, and for positive infinity.

26.8.6.1 Associated Laguerre polynomials

double assoc_laguerre(unsigned n, unsigned m, double x);
float assoc_laguerref(unsigned n, unsigned m, float x);

254) 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 $m \geq 128$.

### 26.8.6.2 Associated Legendre functions

```c
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:

$$P_m^l(x) = \frac{1 - x^2} {m!} \frac{d^m}{dx^m} P_l(x), \quad \text{for } |x| \leq 1,$$

where $l$ is $l$, $m$ is $m$, and $x$ is $x$.

Remarks: The effect of calling each of these functions is implementation-defined if $l \geq 128$.

### 26.8.6.3 Beta function

```c
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 $y$.

### 26.8.6.4 Complete elliptic integral of the first kind

```c
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 26.8.6.11.

### 26.8.6.5 Complete elliptic integral of the second kind

```c
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 26.8.6.12.
26.8.6.6 Complete elliptic integral of the third kind

```c
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);
```

**Effects:** These functions compute the complete elliptic integral of the third kind of their respective arguments \( k \) and \( \nu \).

**Returns:**
\[ \Pi(\nu, k) = \Pi(\nu, k, \pi/2) \], for \(|k| \leq 1\),

where \( k \) is \( k \) and \( \nu \) is \( \nu \).

See also 26.8.6.13.

26.8.6.7 Regular modified cylindrical Bessel functions

```c
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);
```

**Effects:** These functions compute the regular modified cylindrical Bessel functions of their respective arguments \( \nu \) and \( x \).

**Returns:**
\[ I_\nu(x) = i^{-\nu}J_\nu(ix) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k! \Gamma(\nu + k + 1)}, \text{ for } x \geq 0, \]

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 26.8.6.8.

26.8.6.8 Cylindrical Bessel functions of the first kind

```c
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);
```

**Effects:** These functions compute the cylindrical Bessel functions of the first kind of their respective arguments \( \nu \) and \( x \).

**Returns:**
\[ J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k(x/2)^{\nu+2k}}{k! \Gamma(\nu + k + 1)}, \text{ for } x \geq 0, \]

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.

26.8.6.9 Irregular modified cylindrical Bessel functions

```c
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);
```

**Effects:** These functions compute the irregular modified cylindrical Bessel functions of their respective arguments \( \nu \) and \( x \).

**Returns:**
\[ K_\nu(x) = (\pi/2)^{\nu+1}(J_\nu(ix) + iN_\nu(ix)) = \begin{cases} \frac{\pi}{2} \frac{I_{-\nu}(x) - I_\nu(x)}{\sin \nu \pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\ \frac{\pi}{2} \lim_{\mu \to \nu} \frac{I_{-\mu}(x) - I_\mu(x)}{\sin \mu \pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases} \]

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 26.8.6.7, 26.8.6.8, 26.8.6.10.
26.8.6.10 Cylindrical Neumann functions

**double** cyl_neumann(double nu, double x);
**float** cyl_neumannf(float nu, float x);
**long double** 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` >= 128.

See also 26.8.6.8.

26.8.6.11 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);

*Effects:* These functions compute the incomplete elliptic integral of the first kind of their respective arguments `k` and `phi` (\( \phi \) measured in radians).

*Returns:* 

\[ F(k, \phi) = \int_0^\phi \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}}, \text{ for } |k| \leq 1, \]

where \( k \) is `k` and \( \phi \) is `phi`.

26.8.6.12 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);

*Effects:* These functions compute the incomplete elliptic integral of the second kind of their respective arguments `k` and `phi` (\( \phi \) measured in radians).

*Returns:* 

\[ E(k, \phi) = \int_0^\phi \sqrt{1 - k^2 \sin^2 \theta} d\theta, \text{ for } |k| \leq 1, \]

where \( k \) is `k` and \( \phi \) is `phi`.

26.8.6.13 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);

*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:* 

\[ \Pi(\nu, k, \phi) = \int_0^\phi \frac{d\theta}{(1 - \nu \sin^2 \theta)\sqrt{1 - k^2 \sin^2 \theta}}, \text{ for } |k| \leq 1, \]

where \( \nu \) is `nu`, \( k \) is `k`, and \( \phi \) is `phi`.
26.8.6.14 Exponential integral [sf.cmath.expint]

double expint(double x);
float expintf(float x);
long double expintl(long double x);

1 Effects: These functions compute the exponential integral of their respective arguments x.
2

\[
Ei(x) = - \int_{-x}^{\infty} \frac{e^{-t}}{t} \, dt
\]

where \( x \) is x.

26.8.6.15 Hermite polynomials [sf.cmath.hermite]

double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

1 Effects: These functions compute the Hermite polynomials of their respective arguments \( n \) and \( x \).
2

\[
H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}
\]

where \( n \) is \( n \) and \( x \) is \( x \).
3 Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

26.8.6.16 Laguerre polynomials [sf.cmath.laguerre]

double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

1 Effects: These functions compute the Laguerre polynomials of their respective arguments \( n \) and \( x \).
2

\[
L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x}) , \quad \text{for } x \geq 0,
\]

where \( n \) is \( n \) and \( x \) is \( x \).
3 Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

26.8.6.17 Legendre polynomials [sf.cmath.legendre]

double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

1 Effects: These functions compute the Legendre polynomials of their respective arguments \( l \) and \( x \).
2

\[
\mathcal{P}_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l , \quad \text{for } |x| \leq 1,
\]

where \( l \) is 1 and \( x \) is \( x \).
3 Remarks: The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).

26.8.6.18 Riemann zeta function [sf.cmath.riemann.zeta]

double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double x);

1 Effects: These functions compute the Riemann zeta function of their respective arguments \( x \).
Returns:
\[
\zeta(x) = \begin{cases} 
\sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\
\frac{1}{1 - 2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \leq x \leq 1 \\
2^x \pi^{x-1} \sin\left(\frac{\pi x}{2}\right) \Gamma(1-x) \zeta(1-x), & \text{for } x < 0
\end{cases}
\]
where \(x\) is \(x\).

26.8.6.19 Spherical Bessel functions of the first kind

\[
double\ sph\_bessel(unsigned\ n,\ double\ x);
float\\ sph\_besself(unsigned\ n,\ float\ x);
long\ double\ sph\_bessell(unsigned\ n,\ long\ double\ x);
\]

Effects: These functions compute the spherical Bessel functions of the first kind of their respective arguments \(n\) and \(x\).

2 Returns:
\[
j_n(x) = (\pi/2x)^{1/2}J_{n+1/2}(x), \quad \text{for } x \geq 0,
\]
where \(n\) is \(n\) and \(x\) is \(x\).

3 Remarks: The effect of calling each of these functions is implementation-defined if \(n \geq 128\).
See also 26.8.6.8.

26.8.6.20 Spherical associated Legendre functions

\[
double\ sph\_legendre(unsigned\ l,\ unsigned\ m,\ double\ theta);
float\ sph\_legendref(unsigned\ l,\ unsigned\ m,\ float\ theta);
long\ double\ sph\_legendrel(unsigned\ l,\ unsigned\ m,\ long\ double\ theta);
\]

Effects: These functions compute the spherical associated Legendre functions of their respective arguments \(l\), \(m\), and \(theta\) (\(theta\) measured in radians).

2 Returns:
\[
Y_{l}^{m}(\theta,0)
\]
where
\[
Y_{l}^{m}(\theta,\phi) = (-1)^m \left[\frac{(2l + 1)(l - m)!}{4\pi (l + m)!}\right]^{1/2} P_{l}^{m}(\cos\theta) e^{im\phi}, \quad \text{for } |m| \leq l,
\]
and \(l\) is \(l\), \(m\) is \(m\), and \(theta\) is \(theta\).

3 Remarks: The effect of calling each of these functions is implementation-defined if \(l \geq 128\).
See also 26.8.6.2.

26.8.6.21 Spherical Neumann functions

\[
double\ sph\_neumann(unsigned\ n,\ double\ x);
float\ sph\_neumannf(unsigned\ n,\ float\ x);
long\ double\ sph\_neumannl(unsigned\ n,\ long\ double\ x);
\]

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 \(x\).

2 Returns:
\[
n_n(x) = (\pi/2x)^{1/2}N_{n+1/2}(x), \quad \text{for } x \geq 0,
\]
where \(n\) is \(n\) and \(x\) is \(x\).

3 Remarks: The effect of calling each of these functions is implementation-defined if \(n \geq 128\).
See also 26.8.6.10.
26.9 Numbers

26.9.1 Header <numbers> synopsis

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;
   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>;
}

26.9.2 Mathematical constants

1 The library-defined partial specializations of mathematical constant variable templates are initialized with the nearest representable values of e, log2 e, log10 e, π, 1/π, ln 2, ln 10, √2, √3, 1/√3, the Euler-Mascheroni γ constant, and the golden ratio φ constant 1+√5/2, respectively.

2 Pursuant to 16.5.4.2.1, a program may partially or explicitly specialize a mathematical constant variable template provided that the specialization depends on a program-defined type.

3 A program that instantiates a primary template of a mathematical constant variable template is ill-formed.
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 Cpp17Clock requirements</td>
<td></td>
</tr>
<tr>
<td>27.4 Time-related traits</td>
<td>&lt;chrono&gt;</td>
</tr>
<tr>
<td>27.5 Class template duration</td>
<td></td>
</tr>
<tr>
<td>27.6 Class template time_point</td>
<td></td>
</tr>
<tr>
<td>27.7 Clocks</td>
<td></td>
</tr>
<tr>
<td>27.8 Civil calendar</td>
<td></td>
</tr>
<tr>
<td>27.9 Class template hh_mm_ss</td>
<td></td>
</tr>
<tr>
<td>27.10 12/24 hour functions</td>
<td></td>
</tr>
<tr>
<td>27.11 Time zones</td>
<td></td>
</tr>
<tr>
<td>27.12 Formatting</td>
<td></td>
</tr>
<tr>
<td>27.13 Parsing</td>
<td></td>
</tr>
<tr>
<td>27.14 C library time utilities</td>
<td>&lt;ctime&gt;</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

```cpp
#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, 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;
```
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);

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);

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const duration<Rep1, Period>& d, const Rep2& s);

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const Rep1& s, const duration<Rep2, Period>& d);

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator/(const duration<Rep1, Period>& d, const Rep2& s);

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);

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

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);

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<bits::signed integer type of at least 64 bits, nano>>;
using microseconds = duration<bits::signed integer type of at least 55 bits, micro>>;
using milliseconds = duration<bits::signed integer type of at least 45 bits, milli>>;
using seconds = duration<bits::signed integer type of at least 35 bits,>>;
using minutes = duration<bits::signed integer type of at least 29 bits, ratio<60>>;
using hours = duration<bits::signed integer type of at least 23 bits, ratio<3600>>;
using days = duration<bits::signed integer type of at least 25 bits, ratio_multiply<ratio<24>, hours::period>>;
using weeks = duration<bits::signed integer type of at least 22 bits, ratio_multiply<ratio<7>, days::period>>;
using years = duration<bits::signed integer type of at least 17 bits, ratio_multiply<ratio<146097, 400>, days::period>>;
using months = duration<bits::signed integer type of at least 20 bits, ratio_divide<years::period, ratio<12>>;}

// 27.6.5, 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);

// 27.6.6, 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);
  template<class Clock, class Duration1, three_way_comparable_with<Duration1> Duration2>
  constexpr auto operator<=>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// 27.6.7, 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);

// 27.5.9, specialized algorithms

template<
class Rep, class Period>
  constexpr duration<Rep, Period> abs(duration<Rep, Period> d);

// 27.7.1, class system_clock

class system_clock;

using sys_time = time_point<system_clock, Duration>;
using sys_seconds = sys_time<seconds>;
using sys_days = sys_time<days>;

// 27.7.2, class utc_clock

class utc_clock;

using utc_time = time_point<utc_clock, Duration>;
using utc_seconds = utc_time<seconds>;

// 27.7.3, class tai_clock

class tai_clock;

struct leap_second_info;

template<
class Duration>
  leap_second_info get_leap_second_info(const utc_time<Duration>& ut);

§ 27.2
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);

// 27.7.4, class gps_clock
class gps_clock;

template<class Duration>
    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);

// 27.7.5, 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>& tp);

// 27.7.6, class steady_clock
class steady_clock;

// 27.7.7, class high_resolution_clock
class high_resolution_clock;

// 27.7.8, local time
struct local_t {};

template<class Duration>
    using local_time = time_point<local_t, Duration>;
using local_seconds = local_time<seconds>;
using local_days = local_time<days>;

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const local_time<Duration>& tp);

§ 27.2
// 27.7.9, 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;
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;

// 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_day_last& mdl);

// 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& mwd);

// 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;
constexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
constexpr year_month operator+(const months& dm, const year_month& ym) noexcept;
constexpr year_month operator-(const year_month& ym, const months& dm) noexcept;
constexpr months operator-(const year_month& x, const year_month& y) noexcept;
constexpr year_month operator+(const year_month& ym, const years& dy) noexcept;
constexpr year_month operator+(const years& dy, const year_month& ym) noexcept;
constexpr 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;

constexpr year_month_day operator+(const year_month_day& ymd, const months& dm) noexcept;
constexpr year_month_day operator+(const months& dm, const year_month_day& ymd) noexcept;
constexpr year_month_day operator+(const year_month_day& ymd, const years& dy) noexcept;
constexpr year_month_day operator+(const years& dy, const year_month_day& ymd) noexcept;
constexpr year_month_day operator-(const year_month_day& ymd, const months& dm) noexcept;
constexpr year_month_day operator-(const year_month_day& ymd, const years& dy) noexcept;
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;

cnstexpr bool operator==(const year_month_day_last& x,
                        const year_month_day_last& y) noexcept;
cnstexpr strong_ordering operator<=>(const year_month_day_last& x,
                                     const year_month_day_last& y) noexcept;

cnstexpr year_month_day_last
  operator+(const year_month_day_last& ymdl, const months& dm) noexcept;
cnstexpr year_month_day_last
  operator+(const months& dm, const year_month_day_last& ymdl) noexcept;
cnstexpr year_month_day_last
  operator+(const year_month_day_last& ymdl, const years& dy) noexcept;
cnstexpr year_month_day_last
  operator+(const years& dy, const year_month_day_last& ymdl) 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;

cnstexpr bool operator==(const year_month_weekday& x,
                        const year_month_weekday& y) noexcept;

cnstexpr year_month_weekday
  operator+(const year_month_weekday& ymwd, const months& dm) noexcept;
cnstexpr year_month_weekday
  operator+(const months& dm, const year_month_weekday& ymwd) noexcept;
cnstexpr year_month_weekday
  operator+(const year_month_weekday& ymwd, const years& dy) noexcept;
cnstexpr year_month_weekday
  operator+(const years& dy, const year_month_weekday& ymwd) 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;

cnstexpr 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);

// 27.8.18, civil calendar conventional syntax operators
constexpr year_month
operator/(const year& y, const month& m) noexcept;
constexpr year_month
operator/(const year& y, int m) noexcept;
constexpr month_day
operator/(const month& m, const day& d) noexcept;
constexpr month_day
operator/(const month& m, int d) noexcept;
constexpr month_day
operator/(int m, const day& d) noexcept;
constexpr month_day
operator/(const day& d, const month& m) noexcept;
constexpr month_day
operator/(const day& d, int m) noexcept;
constexpr month_weekday_last
operator/(const month& m, const weekday_last& wdl) noexcept;
constexpr month_weekday_last
operator/(int m, const weekday_last& wdl) noexcept;
constexpr month_weekday_last
operator/(const weekday_last& wdl, const month& m) noexcept;
constexpr month_weekday_last
operator/(const weekday_last& wdl, int m) noexcept;

§ 27.2
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_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
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const sys_info& si);

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);

§ 27.2 1236
// 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

namespace chrono { 
  // 27.13, parsing
  template<class charT, class traits, class Alloc, class Parsable>
  unspecified
  parse(const basic_string<charT, traits, Alloc>& format, Parsable& tp);

  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);
}
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 literals {
    inline namespace chrono_literals {
        // 27.5.8, 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: This means C1 did not wrap around between t1 and t2. —end note]

<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>

[Note: 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]

4 A type TC meets the Cpp17TrivialClock requirements if:

(4.1) — TC meets the Cpp17Clock requirements (27.3),
(4.2) — 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: This means, in particular, that operations on these types will not throw exceptions. —end note]
(4.3) — lvalues of the types TC::rep, TC::duration, and TC::time_point are swappable (16.5.3.2),
(4.4) — the function TC::now() does not throw exceptions, and
(4.5) — 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> { };
27.4.2 duration_values

```cpp
template<class Rep>
struct duration_values {
  public:
    static constexpr Rep zero() noexcept;
    static constexpr Rep min() noexcept;
    static constexpr Rep max() noexcept;
};
```

The duration template uses the duration_values trait to construct special values of the duration’s representation (Rep). This is done because the representation might be a class type with behavior which 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.

```cpp
static constexpr Rep zero() noexcept;
```

Returns: Rep(0). [Note: Rep(0) is specified instead of Rep() because Rep() may have some other meaning, such as an uninitialized value. — end note]

Remarks: The value returned shall be the additive identity.

```cpp
static constexpr Rep min() noexcept;
```

Returns: numeric_limits<Rep>::lowest().

Remarks: The value returned shall compare less than or equal to zero().

```cpp
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

```cpp
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: 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]

```cpp
[Note: The typedef name type is a synonym for the duration with the largest tick period possible where both duration arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion with no truncation error, although floating-point durations may have round-off errors. — end note]
```

```cpp
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

```cpp
template<class T> struct is_clock;
```

is_clock is a Cpp17UnaryTypeTrait (20.15.1) 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:
© ISO/IEC  

— the qualified-ids `T::rep`, `T::period`, `T::duration`, and `T::time_point` are valid and each denotes a type (13.10.2),

— the expression `T::is_steady` is well-formed when treated as an unevaluated operand,

— 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

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:
            // 27.5.1, 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& operator=(const duration&);

            // 27.5.2, observer
            constexpr rep count() const;

            // 27.5.3, arithmetic
            constexpr common_type_t<duration> operator+() const;
            constexpr common_type_t<duration> operator-() 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 duration& rhs);

            // 27.5.4, 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.

3 If `Period` is not a specialization of `ratio`, the program is ill-formed. If `Period::num` is not positive, the program is ill-formed.

§ 27.5 1241
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:
```
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]

### 27.5.1 Constructors

```
template<class Rep2>
constexpr explicit duration(const Rep2& r);
```

**Constraints:**
- `is_convertible_v<const Rep2&, rep>` is true
- (1.1) `treat_as_floating_point_v<rep>` is true or
- (1.2) `treat_as_floating_point_v<Rep2>` is false.  

[Example:
```
duration<int, milli> d(3);  // OK
duration<int, milli> d(3.5);  // error
```
— end example]

**Postconditions:** `count() == static_cast<rep>(r)`.

```
template<class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& d);
```

**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: 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:
```
duration<int, milli> ms(3);  // OK
duration<int, micro> us = ms;  // error
```
— end example]

**Effects:** Initializes `rep_` with `duration_cast<duration>(d).count()`.

### 27.5.2 Observer

```
constexpr rep count() const;
```

**Returns:** `rep_`.

### 27.5.3 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.

27.5.4 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()).

27.5.5 Non-member arithmetic

In the function descriptions that follow, unless stated otherwise, let CD represent the return type of the function.

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);
   Returns: CD(CD(lhs).count() + CD(rhs).count()).
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);

3 Returns: CD(CD(lhs).count() - CD(rhs).count()).

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const duration<Rep1, Period>& d, const Rep2& s);

4 Constraints: is_convertible_v<const Rep2&, common_type_t<Rep1, Rep2>> is true.

5 Returns: CD(CD(d).count() * s).

template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const Rep1& s, const duration<Rep2, Period>& d);

6 Constraints: is_convertible_v<const Rep1&, common_type_t<Rep1, Rep2>> is true.

7 Returns: d * s.

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr duration<common_type_t<Rep1, Period1>, duration<Rep2, Period2>>
operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Let CD be common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>.

8 Returns: CD(CD(lhs).count() / CD(rhs).count()).

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);

9 Returns: CD(CD(lhs).count() % CD(rhs).count()).

27.5.6 Comparisons [time.duration.comparisons]

In the function descriptions that follow, CT represents common_type_t<A, B>, where A and B are the types of the two arguments to the function.

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

2 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);

3 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);
4
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);
5
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);
6
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);
7
Returns: CT(lhs).count() <=> CT(rhs).count().

§ 27.5.7 Conversions

template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);
1
Constraints: ToDuration is a specialization of duration.
2
Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_type<
type;typename ToDuration::rep, Rep, intmax_t>::type.
(2.1) — If CF::num == 1 and CF::den == 1, returns
ToDuration(static_cast<
type;typename ToDuration::rep>(d.count()))
(2.2) — otherwise, if CF::num != 1 and CF::den == 1, returns
ToDuration(static_cast<
type;typename ToDuration::rep>(
static_cast<
type;CR>(d.count())) * static_cast<
type;CR>(CF::num)))
(2.3) — otherwise, if CF::num == 1 and CF::den != 1, returns
ToDuration(static_cast<
type;typename ToDuration::rep>(
static_cast<
type;CR>(d.count())) / static_cast<
type;CR>(CF::den)))
(2.4) — otherwise, returns
ToDuration(static_cast<
type;typename ToDuration::rep>(
static_cast<
type;CR>(d.count())) * static_cast<
type;CR>(CF::num) / static_cast<
type;CR>(CF::den)))
3
[Note: 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);
4
Constraints: ToDuration is a specialization of duration.
5
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);
6
Constraints: ToDuration is a specialization of duration.
7
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.

### 27.5.8 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: 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 hours.

Returns: A duration literal representing minutes minutes.

Returns: A duration literal representing sec seconds.

[Note: 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.

### 27.5.9 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.
27.5.10 I/O

template<class charT, class traits, class Rep, class Period>

operator<<(basic_ostream<charT, traits>& os, const duration<Rep, Period>& d);

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:

(1.1) — If Period::type is atto, units-suffix is "as".
(1.2) — Otherwise, if Period::type is femto, units-suffix is "fs".
(1.3) — Otherwise, if Period::type is pico, units-suffix is "ps".
(1.4) — Otherwise, if Period::type is nano, units-suffix is "ns".
(1.5) — Otherwise, if Period::type is micro, it is implementation-defined whether units-suffix is "µs" ("\u00b5s\u0073") or "us".
(1.6) — Otherwise, if Period::type is milli, units-suffix is "ms".
(1.7) — Otherwise, if Period::type is centi, units-suffix is "cs".
(1.8) — Otherwise, if Period::type is deci, units-suffix is "ds".
(1.9) — Otherwise, if Period::type is ratio<1>, units-suffix is "s".
(1.10) — Otherwise, if Period::type is deca, units-suffix is "das".
(1.11) — Otherwise, if Period::type is hecto, units-suffix is "hs".
(1.12) — Otherwise, if Period::type is kilo, units-suffix is "ks".
(1.13) — Otherwise, if Period::type is mega, units-suffix is "Ms".
(1.14) — Otherwise, if Period::type is giga, units-suffix is "Gs".
(1.15) — Otherwise, if Period::type is tera, units-suffix is "Ts".
(1.16) — Otherwise, if Period::type is peta, units-suffix is "Ps".
(1.17) — Otherwise, if Period::type is exa, units-suffix is "Es".
(1.18) — Otherwise, if Period::type is ratio<60>, units-suffix is "min".
(1.19) — Otherwise, if Period::type is ratio<3600>, units-suffix is "h".
(1.20) — Otherwise, if Period::type is ratio<86400>, units-suffix is "d".
(1.21) — Otherwise, if Period::type::den == 1, units-suffix is "[num]s".
(1.22) — 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.

Returns: os.

template<class charT, class traits, class Rep, class Period, class Alloc = allocator<charT>>

to_stream(basic_istream<charT, traits>& is, const charT* fmt, duration<Rep, Period>& d, basic_string<charT, traits, Alloc>* abbrev, nullptrptr, minutes* offset = nullptrptr);

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: is.

27.6 Class template time_point

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:
    // 27.6.1, 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);

    // 27.6.2, observer
    constexpr duration time_since_epoch() const;

    // 27.6.3, 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);

    // 27.6.4, special values
    static constexpr time_point min() noexcept;
    static constexpr time_point max() noexcept;
  };
}

1 Clock shall either meet the Cpp17Clock requirements (27.3) or be the type local_t.

2 If Duration is not an instance of duration, the program is ill-formed.

27.6.1 Constructors

constexpr time_point();

1 Effects: Initializes d_ with duration::zero(). Such a time_point object represents the epoch.

constexpr explicit time_point(const duration& d);

2 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);

3 Constraints: is_convertible_v<Duration2, duration> is true.

4 Effects: Initializes d_ with t.time_since_epoch().

27.6.2 Observer

constexpr duration time_since_epoch() const;

1 Returns: d_.
27.6.3 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_++);
```

```cpp
constexpr time_point& operator--();
Effects: Equivalent to: --d_.
Returns: *this.
```

```cpp
constexpr time_point operator--(int);
Effects: Equivalent to: return time_point(d__--);
```

```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.
```

27.6.4 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()).
```

27.6.5 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: CT(lhs.time_since_epoch() + rhs), where 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: rhs + lhs.
```

```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: CT(lhs.time_since_epoch() - rhs), where 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: lhs.time_since_epoch() - rhs.time_since_epoch().
```

27.6.6 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: lhs.time_since_epoch() == rhs.time_since_epoch().

template<class Clock, class Duration1, class Duration2>
constexpr bool operator<(const time_point<Clock, Duration1>& lhs,
                         const time_point<Clock, Duration2>& rhs);
2

Returns: lhs.time_since_epoch() < rhs.time_since_epoch().

template<class Clock, class Duration1, class Duration2>
constexpr bool operator>(const time_point<Clock, Duration1>& lhs,
                         const time_point<Clock, Duration2>& rhs);
3

Returns: rhs < lhs.

template<class Clock, class Duration1, class Duration2>
constexpr bool operator<=(const time_point<Clock, Duration1>& lhs,
                         const time_point<Clock, Duration2>& rhs);
4

Returns: !(rhs < lhs).

template<class Clock, class Duration1, class Duration2>
constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
                         const time_point<Clock, Duration2>& rhs);
5

Returns: !(lhs < rhs).

template<class Clock, class Duration1,
          three_way_comparable_with<Duration1> Duration2>
constexpr auto operator<=>(const time_point<Clock, Duration1>& lhs,
                           const time_point<Clock, Duration2>& rhs);
6

Returns: lhs.time_since_epoch() <=> rhs.time_since_epoch().

27.6.7 Conversions [time.point.cast]

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);
1

Constraints: ToDuration is a specialization of duration.

Returns:
    time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()))

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> floor(const time_point<Clock, Duration>& tp);
2

Constraints: ToDuration is a specialization of duration.

Returns: time_point<Clock, ToDuration>(floor<ToDuration>(tp.time_since_epoch())).

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> ceil(const time_point<Clock, Duration>& tp);
3

Constraints: ToDuration is a specialization of duration.

Returns: time_point<Clock, ToDuration>(ceil<ToDuration>(tp.time_since_epoch())).

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> round(const time_point<Clock, Duration>& tp);
4

Constraints: ToDuration is a specialization of duration, and
treat_as_floating_point_v<typename ToDuration::rep> is false.

Returns: time_point<Clock, ToDuration>(round<ToDuration>(tp.time_since_epoch())).

27.7 Clocks [time.clock]

The types defined in this subclause meet the Cpp17TrivialClock requirements (27.3) unless otherwise specified.
27.7.1 Class `system_clock`

27.7.1.1 Overview

```cpp
namespace std::chrono {
    class system_clock {
        public:
            using rep = see below;
            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:
   ```cpp
   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
   ```

27.7.1.2 Members

```cpp
using system_clock::rep = unspecified;
```

1 Constraints: `system_clock::duration::min() < system_clock::duration::zero()` is true.

   [Note: This implies that `rep` is a signed type. — end note]

```cpp
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.

```cpp
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.

27.7.1.3 Non-member functions

```cpp
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_as_floating_point_v<typename Duration::rep> is false, and Duration{1} < days{1} is true.

2 Effects: Equivalent to:

   ```cpp
   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:

   ```cpp
   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
   ```
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const sys_days& dp);

Effects: os << year_month_day(dp).

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,
        sys_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
        minutes* offset = nullptr);

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.

Returns: is.

27.7.2 Class utc_clock

time.clock.utc

27.7.2.1 Overview

time.clock.utc.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);
    }
}

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: 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:
clock_cast<utc_clock>({sys_seconds(sys_days{1972y/1/1})).time_since_epoch()} is 0s.
clock_cast<utc_clock>({sys_seconds(sys_days{2000y/1/1})).time_since_epoch()} is 946'684'822s, which is 10'957 * 86'400s + 22s.
— end example]

utc_clock is not a Cpp17TrivialClock unless the implementation can guarantee that utc_clock::now() does not propagate an exception. [Note: noexcept(from_sys(system_clock::now())) is false. — end note]

27.7.2.2 Member functions
time.clock.utc.members

static time_point now();

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);

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);

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:
  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;
  u = 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]

27.7.2.3 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);

Effects: Equivalent to:

  return os << format("{:%F %T}", t);

[Example:
  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>>
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 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.

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.

27.7.3 Class tai_clock

27.7.3.1 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: noexcept(from_utc(utc_clock::now())) is false. — end note]

27.7.3.2 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:
utc_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 378691210s

[Note:
378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s
—end note]

template<class Duration>
static 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:
378691210s == sys_days{1970y/January/1} - sys_days{1958y/January/1} + 10s
—end note]

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 tai_time<Duration>& t);

Effects: Equivalent to:
return os << format(STATICALLY-WIDEN<charT>"{:%F %T}", t);

[Example:
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 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.

27.7.4 Class gps_clock

27.7.4.1 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>;

§ 27.7.4.1 1255
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 1958/January/1 to 1980/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.

`gps_clock` is not a `Cpp17TrivialClock` unless the implementation can guarantee that `gps_clock::now()` does not propagate an exception. [Note: `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 `gps_time`.

- **template<class Duration>
  static utc_time<common_type_t<Duration, seconds>>
  to_utc(const gps_time<Duration>& t) noexcept;**
  - Returns: `gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 315964809s`
  - [Note: 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;**
  - Returns: `gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 315964809s`
  - [Note: 315964809s == `sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s`
  — 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 gps_time<Duration>& t);**
  - Effects: Equivalent to:
    ```
    return os << format(STATICALLY-WIDEN<charT>("{:0F %T}\n"), t);
    ```
  - [Example:]
    ```
    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.

27.7.5 Type file_clock

27.7.5.1 Overview

namespace std::chrono {
    using file_clock = see below;
}

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: The type that file_clock denotes may be in a different namespace than std::chrono, such as std::filesystem. —end note]

27.7.5.2 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.

27.7.5.3 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.

27.7.6 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.

27.7.7 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.

27.7.8 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);

Effects: Attempts to parse the input stream is into the local_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.

Returns: is.

27.7.9 time_point conversions

27.7.9.1 Class template clock_time_conversion

namespace std::chrono {
    template<class DestClock, class SourceClock>
    struct clock_time_conversion {};
}

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.

Several specializations are provided by the implementation, as described in 27.7.9.2, 27.7.9.3, 27.7.9.4, and
27.7.9.5.

27.7.9.2 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;

Returns: t.

template<class SystemClock, SystemClock>
struct clock_time_conversion<system_clock, system_clock> {
    template<class Duration>
    sys_time<Duration>
        operator()(const sys_time<Duration>& t) const;
};

Returns: t.

template<class Duration>
    sys_time<Duration>
        operator()(const sys_time<Duration>& t) const;

Returns: t.

template<class UtcClock, UtcClock>
struct clock_time_conversion<utc_clock, utc_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.

27.7.9.3 Conversions between system_clock and utc_clock
[time.clock.cast.sys.utc]

```cpp
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;
};
```

```cpp
template<class Duration>
utc_time<common_type_t<Duration, seconds>>
operator()(const sys_time<Duration>& t) const;

Returns: utc_clock::from_sys(t).
```

```cpp
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;
};
```

```cpp
template<class Duration>
sys_time<common_type_t<Duration, seconds>>
operator()(const utc_time<Duration>& t) const;

Returns: utc_clock::to_sys(t).
```

27.7.9.4 Conversions between system_clock and other clocks
[time.clock.cast.sys]

```cpp
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));
};
```

```cpp
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).
```

```cpp
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));
};
```

```cpp
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).
```
27.7.9.5 Conversions between utc_clock and other clocks

```
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));
};
```

```
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).

```
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));
};
```

```
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).

27.7.9.6 Function template clock_cast

```
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;  

Effects: Initializes d_ with d. The value held is unspecified if d is not in the range [0, 255].

constexpr day& operator++() noexcept;  

Effects: ++d_.

Returns: *this.

constexpr day operator++(int) noexcept;  

Effects: ++(*this).

Returns: A copy of *this as it existed on entry to this member function.

constexpr day& operator--() noexcept;  

Effects: Equivalent to: --d_.

Returns: *this.

constexpr day operator--(int) noexcept;  

Effects: --(*this).

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: x + y.

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)}. 

§ 27.8.3.3
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_.

customexpr bool ok() const noexcept;

Returns: 1 <= m_ && m_ <= 12.

27.8.4.3 Non-member functions

constexpr bool operator==(const month& x, const month& y) noexcept;

Returns: unsigned{x} == unsigned{y}.

constexpr strong_ordering operator<=>(const month& x, const month& y) noexcept;

Returns: unsigned{x} <=> unsigned{y}.

constexpr month operator+(const month& x, const months& y) noexcept;

Returns:
month{modulo(static_cast<long long>(unsigned{x}) + (y.count() - 1), 12) + 1}
where modulo(n, 12) computes the remainder of n divided by 12 using Euclidean division. [Note:
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.ok(). — end note] [Example:
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.ok() == true and y.ok() == true, returns a value m in the range [months{0},
months{11}] satisfying y + m == x. Otherwise the value returned is unspecified. [Example: 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:
return os << (m.ok() ?
    format(os.getloc(), STATICALLY-WIDEN<charT>("{:%b}"), m) :
    format(os.getloc(), STATICALLY-WIDEN<charT>("{} is not a valid month"),
        static_cast<unsigned>(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);

Effects: Attempts to parse the input stream is into the month m using the format flags given in the
NTCTS fmt as specified in 27.13. If the parse fails to decode a valid month, is.setstate(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 \(\text{int}\) 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

1 constexpr explicit year(int y) noexcept;
   
   Effects: Initializes \(y_\) with \(y\). The value held is unspecified if \(y\) is not in the range \([-32767, 32767]\).

2 constexpr year& operator++() noexcept;
   
   Effects: \(\mathbf{++}y_\).

3 Returns: \(*\text{this}.*\)

4 constexpr year operator++(int) noexcept;
   
   Effects: \(\mathbf{++}(\mathbf{\text{this}})\).

5 Returns: A copy of \(*\text{this}\) as it existed on entry to this member function.

6 constexpr year& operator--() noexcept;

7 Returns: \(*\text{this}.*\)

8 constexpr year operator--(int) noexcept;
   
   Effects: \(\mathbf{--}(\mathbf{\text{this}})\).

9 Returns: A copy of \(*\text{this}\) as it existed on entry to this member function.

10 constexpr year& operator+=(const years& y) noexcept;
    
    Effects: \(*\text{this} = *\text{this} + y\).

11 Returns: \(*\text{this}\).
constexpr year& operator-=(const years& y) noexcept;
Effects: \( \ast\text{this} = \ast\text{this} - y \).
Returns: \( \ast\text{this} \).

constexpr year operator+() const noexcept;
Returns: \( \ast\text{this} \).

constexpr year::operator-() const noexcept;
Returns: \( -y \).

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: \( \text{min().y} \leq y \&\& y \leq \text{max().y} \).

static constexpr year min() noexcept;
Returns: \( -32767 \).

static constexpr year max() noexcept;
Returns: \( 32767 \).

27.8.5.3 Non-member functions
[time.cal.year.nonmembers]

constexpr bool operator==(const year& x, const year& y) noexcept;
Returns: \( \text{int(x)} == \text{int(y)} \).

constexpr strong_ordering operator<=>(const year& x, const year& y) noexcept;
Returns: \( \text{int(x)} \<\=>\text{int(y)} \).

constexpr year operator+(const year& x, const years& y) noexcept;
Returns: \( \text{year\{int(x) + y.count\}} \).

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: \( \text{years\{int(x) - int(y)\}} \).

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}\"), 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: 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_.

[Example: If dp represents 1970-01-01, the constructed weekday represents Thursday by storing 4 in wd_. — end example]
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

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: 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
weekday holding a value in the range \([0, 6]\) even if \(!x\.\text{ok}\()\). —end note] [Example: Monday +
days\{6\} == Sunday. —end example]

constexpr weekday operator\+(const days& x, const weekday& y) noexcept;

Returns: \(y + x\).

constexpr weekday operator\-(const weekday& x, const days& y) noexcept;

Returns: \(x + -y\).

constexpr days operator\-(const weekday& x, const weekday& y) noexcept;

Returns: If \(x\.\text{ok}\() == \text{true}\) and \(y\.\text{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. [Example: Sunday - Monday ==
days\{6\}. —end example]

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday& wd);

Effects: Equivalent to:

\[
\text{return os }\ll\text{(wd\.\text{ok}() ? format(os\.getloc(), STATICALLY-WIDEN<charT>("\{:%a\}"), wd) :
format(os\.getloc(), STATICALLY-WIDEN<charT>("\{\} is not a valid weekday"),
static_cast<unsigned>(wd\.wd\_)));}
\]

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);

Effects: Attempts to parse the input stream \(is\) into the \(weekday\) \(wd\) using the format flags given in
the NTCTS \(fmt\) as specified in 27.13. If the parse fails to decode a valid weekday, \(is\.\text{setstate}(\text{ios\_base::failbit})\) is called and \(wd\) 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.

Returns: \(is\).

27.8.7 Class weekday_indexed

27.8.7.1 Overview

namespace std::chrono {

class weekday_indexed {

chrono::weekday wd_; // exposition only
unsigned char index_; // exposition only

public:

weekday_indexed() = default;

constexpr weekday_indexed(const chrono::weekday& wd, unsigned index) noexcept;

constexpr chrono::weekday weekday() const noexcept;
constexpr unsigned index() const noexcept;
constexpr bool ok() const noexcept;

};
}

weekday_indexed represents a 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.

[Note: A weekday_indexed object can be constructed by indexing a weekday with an unsigned. —end note] [Example:

constexpr auto wdi = Sunday[2]; // wdi is the second Sunday of an as yet unspecified month
static_assert(wdi\.weekday() == Sunday);
static_assert(wdi\.index() == 2);]


### 27.8.7.2 Member functions

```cpp
constexpr weekday_indexed(const chrono::weekday& wd, unsigned index) noexcept;
```

1. **Effects:** Initializes \( wd_ \) with \( wd \) and \( index_ \) with \( index \). The values held are unspecified if !\( wd\_\).ok() or \( index \) is not in the range \([0, 7]\).

```cpp
constexpr chrono::weekday weekday() const noexcept;
```

2. **Returns:** \( wd_ \).

```cpp
constexpr unsigned index() const noexcept;
```

3. **Returns:** \( index_ \).

```cpp
constexpr bool ok() const noexcept;
```

4. **Returns:** \( wd\_\).ok() && 1 <= index_ && index_ <= 5.

### 27.8.7.3 Non-member functions

```cpp
constexpr bool operator==(const weekday_indexed& x, const weekday_indexed& y) noexcept;
```

1. **Returns:** \( x\_.\).weekday() == y\_.weekday() && x\_.index() == y\_.index().

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday_indexed& wdi);
```

2. **Effects:** Equivalent to:
   ```cpp
   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

```cpp
namespace std::chrono {
    class weekday_last {
        chrono::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;
    };
}
```

1. **weekday_last** represents the last weekday of a month.

2. **[Note: A weekday_last object can be constructed by indexing a weekday with last. — end note]**

```cpp
constexpr auto wdl = Sunday[last]; // wdl is the last Sunday of an as yet unspecified month
static_assert(wdl.weekday() == Sunday);
```

3. **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;
```

1. **Effects:** Initializes \( wd_ \) with \( wd \).
constexpr chrono::weekday weekday() const noexcept;

Returns: \( wd_ \).

constexpr bool ok() const noexcept;

Returns: \( wd_.ok() \).

### 27.8.8.3 Non-member functions

constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;

Returns: \( x.weekday() == y.weekday() \).

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const weekday_last& wdl);

Effects: Equivalent to:

\[
\text{return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}[last]"), wdl.weekday());}
\]

### 27.8.9 Class \texttt{month\_day}

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;

};
}

\texttt{month\_day} represents a specific day of a specific month, but with an unspecified year. \texttt{month\_day} meets the 
\texttt{Cpp17EqualityComparable} (Table 25) and \texttt{Cpp17LessThanComparable} (Table 26) requirements.

\texttt{month\_day} is a trivially copyable and standard-layout class type.

### 27.8.9.2 Member functions

constexpr month\_day(const chrono::month& m, const chrono::day& d) noexcept;

Effects: Initializes \( m_ \) with \( m \), and \( d_ \) with \( d \).

constexpr chrono::month month() const noexcept;

Returns: \( m_ \).

constexpr chrono::day day() const noexcept;

Returns: \( d_ \).

constexpr bool ok() const noexcept;

Returns: true if \( m_.ok() \) is true, \( 1d <= d_ \), and \( d_ \) is less than or equal to the number of days in \( \text{month} m_ \); otherwise returns false. When \( m_ == \text{February} \), the number of days is considered to be 29.

### 27.8.9.3 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.

## 27.8.10 Class month_day_last

[time.cal.mdlast]

namespace std::chrono {

```cpp
class month_day_last {
    chrono::month m_; // exposition only

    public:
        constexpr explicit month_day_last(const chrono::month& m) noexcept;

        constexpr chrono::month month() const noexcept;
        constexpr bool ok() const noexcept;

        constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;

    };
```

1 month_day_last represents the last day of a month.

2 [Note: 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:

```cpp
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]

3 month_day_last is a trivially copyable and standard-layout class type.

```cpp
constexpr explicit month_day_last(const chrono::month& m) noexcept;
```

Effects: Initializes m_ with m.

```cpp
constexpr month month() const noexcept;
```

Returns: m_.

```cpp
constexpr bool ok() const noexcept;
```

Returns: m_.ok().

```cpp
constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;
```

Returns: x.month() == y.month().

§ 27.8.10
constexpr strong_ordering operator<=>(const month_day_last& x, const month_day_last& y) noexcept;

Returns: \(x.\text{month}() \leftrightarrow y.\text{month}()\).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);

Effects: Equivalent to:

\[
\text{return os } \ll \text{ format(os.getloc(), } \text{STATICALLY-WIDEN}<\text{charT>("{}/last"), mdl.\text{month})};
\]

27.8.11 Class month_weekday

27.8.11.1 Overview

namespace std::chrono {
    class month_weekday {
        chrono::month m_;  // exposition only
        chrono::weekday_indexed wdi_;  // exposition only
    public:
        constexpr month_weekday(chrono::month m, chrono::weekday_indexed wdi) noexcept;

        constexpr chrono::month month() const noexcept;

        constexpr chrono::weekday_indexed weekday_indexed() const noexcept;

        constexpr bool ok() const noexcept;
    };
}

1 month_weekday represents the \(n\)th weekday of a month, of an as yet unspecified year. To do this the month_weekday stores a month and a weekday_indexed.

2 [Example:

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 example"

3 month_weekday is a trivially copyable and standard-layout class type.

27.8.11.2 Member functions

constexpr month_weekday(chrono::month m, chrono::weekday_indexed wdi) noexcept;

Effects: Initializes \(m_\) with \(m\), and \(wdi_\) with \(wdi\).

constexpr chrono::month month() const noexcept;

Returns: \(m_\).

constexpr chrono::weekday_indexed weekday_indexed() const noexcept;

Returns: \(wdi_\).

constexpr bool ok() const noexcept;

Returns: \(m_.\text{ok}() \&\& wdi_.\text{ok}()\).

27.8.11.3 Non-member functions

constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;

Returns: \(x.\text{month}() == y.\text{month}() \&\& x.\text{weekday_indexed}() == y.\text{weekday_indexed}()\).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday& mdl);

Effects: Equivalent to:

\[
\text{return os } \ll \text{ format(os.getloc(), } \text{STATICALLY-WIDEN}<\text{charT>("{}/{}"), mwd.\text{month}, mwd.\text{weekday_indexed})};
\]
27.8.12 Class month_weekday_last

27.8.12.1 Overview

namespace std::chrono {
    class month_weekday_last {
        chronon::month m_; // exposition only
        chronon::weekday_last wdl_; // exposition only
        public:
            constexpr month_weekday_last(const chronon::month& m,
                                          const chronon::weekday_last& wdl) noexcept;
            constexpr chronon::month month() const noexcept;
            constexpr chronon::weekday_last weekday_last() const noexcept;
            constexpr bool ok() const noexcept;
    };
}

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.

[Example:
    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]

month_weekday_last is a trivially copyable and standard-layout class type.

27.8.12.2 Member functions

constexpr month_weekday_last(const chronon::month& m,
                             const chronon::weekday_last& wdl) noexcept;

    Effects: Initializes m_ with m, and wdl_ with wdl.

constexpr chronon::month month() const noexcept;

    Returns: m_.

constexpr chronon::weekday_last weekday_last() const noexcept;

    Returns: wdl_.

constexpr bool ok() const noexcept;

    Returns: m_.ok() && wdl_.ok().

27.8.12.3 Non-member functions

constexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;

    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& mwdl);

    Effects: Equivalent to:
             return os << format(os.getloc(), STATICALLY-WIDEN<charT>("{}/{}"),
                                  mwdl.month(), mwdl.weekday_last());

27.8.13 Class year_month

27.8.13.1 Overview
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

constexpr year_month(const chrono::year& y, const chrono::month& m) noexcept;
    Effects: Initializes y_ with y, and m_ with m.
    Returns: y_.

constexpr chrono::year year() const noexcept;
    Returns: m_.

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 (12.4.3.2).
    Effects: *this = *this + dm.
    Returns: *this.

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 (12.4.3.2).
    Effects: *this = *this - dm.
    Returns: *this.

constexpr year_month& operator+=(const years& dy) noexcept;
    Effects: *this = *this + dy.
    Returns: *this.

constexpr year_month& operator-=(const years& dy) noexcept;
    Effects: *this = *this - dy.
    Returns: *this.

constexpr bool ok() const noexcept;
    Returns: y_.ok() && m_.ok().

27.8.13.3 Non-member functions

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 (12.4.3.2).

Returns: A year_month value z such that z.ok() && z - ym == dm is true.

Complexity: O(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 (12.4.3.2).

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 (12.4.3.2).

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, const 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 {
    
        chronon::year y_;  // exposition only
        chronon::month m_;  // exposition only
        chronon::day d_;    // exposition only

    public:
        year_month_day() = default;
        constexpr year_month_day(const chronon::year& y, const chronon::month& m,
                                      const chronon::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 chronon::year year() const noexcept;
        constexpr chronon::month month() const noexcept;
        constexpr chronon::day day() const noexcept;
        
        constexpr operator sys_days() const noexcept;
        constexpr explicit operator local_days() const noexcept;
        constexpr bool ok() const noexcept;
    
    }; // namespace std::chrono

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: 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 chronon::year& y, const chronon::month& m,
                                      const chronon::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: This conversion from year_month_day_last to year_month_day may 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;
4 Effects: Equivalent to constructing with sys_days(dp.time_since_epoch()).

constexpr year_month_day& operator+=(const months& m) noexcept;
5 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 (12.4.3.2).
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 (12.4.3.2).

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 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, then returns true; otherwise returns false.

27.8.14.3 Non-member functions

constexpr bool operator==(const year_month_day& x, const year_month_day& y) noexcept;

Returns: x.year() == y.year() && x.month() == y.month() && x.day() == y.day().

constexpr strong_ordering operator<=>(const year_month_day& x, const year_month_day& y) noexcept;

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;

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 (12.4.3.2).

Returns: (ymd.year() / ymd.month() + dm) / ymd.day().

[Note: If ymd.day() is in the range [1d, 28d], ok() will return true for the resultant year_month_day. —end note]

constexpr year_month_day operator+(const months& dm, const year_month_day& ymd) 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 (12.4.3.2).

Returns: ymd + dm.

constexpr year_month_day operator-(const year_month_day& ymd, 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 (12.4.3.2).

Returns: ymd + (-dm).

constexpr year_month_day operator+(const year_month_day& ymd, const years& dy) noexcept;

Returns: (ymd.year() + dy) / ymd.month() / ymd.day().

[Note: If ymd.month() is February and ymd.day() is not in the range [1d, 28d], ok() may return false for the resultant year_month_day. —end note]

constexpr year_month_day operator+(const years& dy, const year_month_day& ymd) noexcept;

Returns: ymd + dy.

constexpr year_month_day operator-(const year_month_day& ymd, const years& dy) noexcept;

Returns: ymd + (-dy).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day& ymd);

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);

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.

Returns: is.

27.8.15 Class year_month_day_last [time.cal.ymdlast]
27.8.15.1 Overview [time.cal.ymdlast.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;
};

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: year_month_day_last supports years- and months-oriented arithmetic, but not days-oriented arithmetic. — end note] year_month_day_last meets the Cpp17EqualityComparable (Table 25) and Cpp17LessThanComparable (Table 26) requirements.

27.8.15.2 Member functions

constexpr year_month_day_last(const chrono::year& y,
const chrono::month_day_last& mdl) noexcept;
1 Effect: Initializes y_ with y and mdl_ with mdl.

constexpr year_month_day_last& operator+=(const months& m) noexcept;
2 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 (12.4.3.2).
3 Effect: *this = *this + m.
4 Returns: *this.

constexpr year_month_day_last& operator-=(const months& m) noexcept;
5 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 (12.4.3.2).
6 Effect: *this = *this - m.
7 Returns: *this.

constexpr year_month_day_last& operator+=(const years& y) noexcept;
8 Effect: *this = *this + y.
9 Returns: *this.

constexpr year_month_day_last& operator-=(const years& y) noexcept;
10 Effect: *this = *this - y.
11 Returns: *this.

constexpr chrono::year year() const noexcept;
12 Returns: y_.

constexpr chrono::month month() const noexcept;
13 Returns: mdl_.month().
constexpr chrono::month_day_last month_day_last() const noexcept;

Returns: mdl_.

constexpr 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: This value may be computed on demand. —end note]

constexpr operator sys_days() const noexcept;

Returns: sys_days(year()/month()/day()).

constexpr explicit operator local_days() const noexcept;

Returns: local_days(sys_days(*this).time_since_epoch()).

constexpr 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().

constexpr 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();

constexpr 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 (12.4.3.2).

Returns: (ymdl.year() / ymdl.month() + dm) / last.

constexpr 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 (12.4.3.2).

Returns: ymdl + dm.

constexpr 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 (12.4.3.2).

Returns: ymdl + (-dm).

constexpr year_month_day_last
operator-(const months& dm, const year_month_day_last& ymdl) noexcept;

Returns: {ymdl.year()}*dy, ymdl.month_day_last()}.
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);

Effects: Equivalent to:

return os << format(os.getloc(), STATICALLY-WIDEN<charT>("/{}/{}"),
ymdl.year(), ymdl.month_day_last());

27.8.16 Class year_month_weekday

27.8.16.1 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 year_month_weekday represents a specific year, month, and nth weekday of the month. year_month_weekday is a field-based time point with a resolution of days. [Note: year_month_weekday 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 meets the Cpp17EqualityComparable (Table 25) requirements.

2 year_month_weekday is a trivially copyable and standard-layout class type.

27.8.16.2 Member functions

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.

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.

constexpr explicit year_month_weekday(const local_days& dp) noexcept;

Effects: Equivalent to constructing with sys_days{dp.time_since_epoch()}. 

§ 27.8.16.2
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 (12.4.3.2).

Effects: *this = *this + m.
Returns: *this.

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 (12.4.3.2).

Effects: *this = *this - m.
Returns: *this.

constexpr year_month_weekday& operator+=(const years& y) noexcept;

Effects: *this = *this + y.
Returns: *this.

constexpr year_month_weekday& 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: wdi_.weekday().

constexpr unsigned index() const noexcept;

Returns: wdi_.index().

constexpr chrono::weekday_indexed weekday_indexed() const noexcept;

Returns: wdi_.

constexpr chrono::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()}.

constexpr bool ok() const noexcept;

Returns: If any of y_.ok(), m_.ok(), or wdi_.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 (12.4.3.2).

Returns: \((ymwd.year() / ymwd.month() + dm) / ymwd.weekday_indexed())\).

constexpr 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 (12.4.3.2).

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 (12.4.3.2).

Returns: ymwd + (-dm).

castexpr year_month_weekday operator+(const months& dm, const year_month_weekday& ymwd) noexcept;

Returns: \(ymwd + dm\).

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: \(\{ymwd.year()+dy, ymwd.month(), ymwd.weekday_indexed()\}\).

template<class charT, class traits>

basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

Effects: Equivalent to:

\[
\text{return os } \ll \text{format(os.getloc(), STATICALLY-WIDEN}<\text{charT}(\"/{}//{}/\{}\)),
\text{ymwd.year()}, \text{ymwd.month()}, \text{ymwd.weekday_indexed()});
\]

27.8.17 Class year_month_weekday_last

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:
    constexpr year_month_weekday_last(const chrono::year& y, const chrono::month& m, const chrono::weekday_last& wdl) noexcept;

    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: `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`.
```

```cpp
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` (12.4.3.2).
Effects: *this = *this + m.
Returns: *this.
```

```cpp
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` (12.4.3.2).
Effects: *this = *this - m.
Returns: *this.
```

```cpp
constexpr year_month_weekday_last& operator+=(const years& y) noexcept;
Effects: *this = *this + y.
Returns: *this.
```

```cpp
constexpr year_month_weekday_last& 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: `wdl_.weekday()`.
```

```cpp
constexpr chrono::weekday_last weekday_last() const noexcept;
Returns: `wdl_`.
```

```cpp
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.
```

```cpp
constexpr explicit operator local_days() const noexcept;
Returns: `local_days{sys_days{*this}.time_since_epoch()}`.
```

```cpp
constexpr bool ok() const noexcept;
Returns: `y_.ok() && m_.ok() && wdl_.ok()`.
```
27.8.17.3 Non-member functions

```cpp
constexpr bool operator==(const year_month_weekday_last& x,
    const year_month_weekday_last& y) noexcept;
```

Returns:

\[ x.\text{year}() == y.\text{year}() && x.\text{month}() == y.\text{month}() && x.\text{weekday}\text{last}() == y.\text{weekday}\text{last}() \]

```cpp
constexpr year_month_weekday_last
    operator+(const year_month_weekday_last& ymwdl, 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` (12.4.3.2).

Returns:

\[ (\text{ymwdl.year}() / \text{ymwdl.month}() + \text{dm}) / \text{ymwdl.weekday}\text{last}() \]

```cpp
constexpr year_month_weekday_last
    operator+(const months& dm, const year_month_weekday_last& ymwdl) 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` (12.4.3.2).

Returns:

\[ \text{ymwdl} + \text{dm}. \]

```cpp
constexpr year_month_weekday_last
    operator+(const year_month_weekday_last& ymwdl, 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` (12.4.3.2).

Returns:

\[ \text{ymwdl} + (-\text{dm}) \]

```cpp
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwdl);
```

Effects: Equivalent to:

\[ \text{return } \text{os} \ll \text{format( os.getloc(), } \text{STATICALLY-WIDEN} \text{<charT>}("{}\slash{}\slash{}"),
    \text{ymwdl.year}(), \text{ymwdl.month}(), \text{ymwdl.weekday}\text{last}()); } \]

27.8.18 Conventional syntax operators

A set of overloaded `operator/` functions provides a conventional syntax for the creation of civil calendar dates.

[Note: 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: 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:

```cpp
year_month ym = 2015y/April;
month_day md1 = April/4;
month_day md2 = 4d/April;
```
— end note]

[Example:

```cpp
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]

```cpp
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& y, const month_day& md) noexcept;

Returns: y / md.month() / md.day().

constexpr year_month_day
operator/(int y, const month_day& md) noexcept;

Returns: year(y) / md.

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()}\}. 

\§ 27.8.18
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.

constexpr year_month_weekday_last
operator/(const month_weekday_last& mwdl, int y) noexcept;

Returns: year(y) / mwdl.

27.9 Class template hh_mm_ss
27.9.1 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
};

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.

If Duration is not an instance of duration, the program is ill-formed.

### 27.9.2 Members

static constexpr unsigned fractional_width = see below;

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: 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>
<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>
using precision = see below;

```cpp
constexpr explicit hh_mm_ss(Duration d);
```

**Effects:** Constructs an object of type `hh_mm_ss` which represents the `Duration` `d` with precision `precision`.

1. Initializes `is_neg` with `d < Duration::zero()`.
2. Initializes `h` with `duration_cast<chrono::hours>(abs(d))`.
3. Initializes `m` with `duration_cast<chrono::minutes>(abs(d) - hours())`.
4. Initializes `s` with `duration_cast<chrono::seconds>(abs(d) - hours() - minutes())`.
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: When `precision::rep` is integral and `precision::period` is `ratio<1>`, `subseconds()` always returns a value equal to 0s. — end note]

**Postconditions:** If `treat_as_floating_point_v<precision::rep>` is true, `to_duration()` returns `d`, otherwise `to_duration()` returns `duration_cast<precision>(d)`.

```cpp
constexpr bool is_negative() const noexcept;
```

**Returns:** `is_neg`.

```cpp
constexpr chrono::hours hours() const noexcept;
```

**Returns:** `h`.

```cpp
constexpr chrono::minutes minutes() const noexcept;
```

**Returns:** `m`.

```cpp
constexpr chrono::seconds seconds() const noexcept;
```

**Returns:** `s`.

```cpp
constexpr precision subseconds() const noexcept;
```

**Returns:** `ss`.

```cpp
constexpr precision to_duration() const noexcept;
```

**Returns:** If `is_neg`, returns `-(h + m + s + ss)`, otherwise returns `h + m + s + ss`.

```cpp
constexpr explicit operator precision() const noexcept;
```

**Returns:** `to_duration()`.

### 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:**

```cpp
for (auto ms : {-4083007ms, 4083007ms, 65745123ms}) {
    hh_mm_ss hms{ms};
    cout << hms << 'n';
}
```

```cpp
cout << hh_mm_ss(65745s) << '\n';
```
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.

constexpr bool is_pm(const hours& h) noexcept;
Returns: 12h <= h && h <= 23h.

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.

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

```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.

```cpp
const time_zone* locate_zone(string_view tz_name) const;
Returns:
(2.1) If `zones` contains an element `tz` for which `tz.name() == tz_name`, a pointer to `tz`;
(2.2) 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()`.

[Note: A `time_zone_link` specifies an alternative name for a `time_zone`. — end note]
```

Throws: If a `const time_zone*` cannot be found as described in the `Returns:` clause, throws a `runtime_error`. [Note: On non-exceptional return, the return value is always a pointer to a valid `time_zone`. — 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

```cpp
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. [Note: 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()`. — 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`.

```cpp
const tzdb& front() const noexcept;
```

Returns: A reference to the first `tzdb` in the container.

```cpp
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`.

Returns: An iterator pointing to the element following the one that was erased, or `end()` if no such element exists.

Postconditions: No pointers, references, or iterators are invalidated except those referring to the erased `tzdb`. [Note: It is not possible to erase the `tzdb` referred to by `begin()`. — end note]

Throws: Nothing.

```cpp
const_iterator begin() const noexcept;
```

Returns: An iterator referring to the first `tzdb` in the container.

```cpp
const_iterator end() const noexcept;
```

Returns: An iterator referring to the position one past the last `tzdb` in the container.

```cpp
const_iterator cbegin() const noexcept;
```

Returns: `begin()`.
const_iterator cend() const noexcept;

Returns: end().

27.11.2.3 Time zone database access

tzdb_list& get_tzdb_list();

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.

Synchronization: It is safe to call this function from multiple threads at one time.

Returns: A reference to the database.

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();

Returns: get_tzdb_list().front().

const time_zone* locate_zone(string_view tz_name);

Returns: get_tzdb().locate_zone(tz_name).

[Note: The time zone database will be initialized if this is the first reference to the database. — end note]

const time_zone* current_zone();

Returns: get_tzdb().current_zone().

27.11.2.4 Remote time zone database support

const tzdb& reload_tzdb();

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().

Synchronization: This function is thread-safe with respect to get_tzdb_list().front() and get_tzdb_list().erase_after().

Postconditions: No pointers, references, or iterators are invalidated.

Returns: get_tzdb_list().front().

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: 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 \texttt{local\_time} to a \texttt{sys\_time} without specifying \texttt{choose::earliest} or \texttt{choose::latest}.

\begin{verbatim}
template<class Duration>
nonexistent\_local\_time(const local\_time<Duration>& tp, const local\_info& i);
\end{verbatim}

\textbf{Preconditions:} \texttt{i.result == local\_info::nonexistent} is true.

\textbf{Effects:} Initializes the base class with a sequence of \texttt{char} equivalent to that produced by \texttt{os.str()} initialized as shown below:

\begin{verbatim}
ostringstream os;
  os << tp << " is in a gap between\n"  << local\_seconds{i.first.end.time\_since\_epoch()} + i.first.offset << ', '\n"  << i.first.abbrev << " and\n"  << local\_seconds{i.second.begin.time\_since\_epoch()} + i.second.offset << ', '\n"  << i.second.abbrev
  " which are both equivalent to\n"  << i.first.end << " UTC";
\end{verbatim}

\begin{example}
#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';
  }
}
\end{example}

Produces the output:

\begin{center}
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{center}

--- end example]

27.11.3.2 Class ambiguous\_local\_time \[time.zone.exception.ambig\]

\begin{verbatim}
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);
    };}
\end{verbatim}

\begin{verbatim}
ambiguous\_local\_time is thrown when an attempt is made to convert an ambiguous \texttt{local\_time} to a \texttt{sys\_time} without specifying \texttt{choose::earliest} or \texttt{choose::latest}.
\end{verbatim}

\begin{verbatim}
template<class Duration>
ambiguous\_local\_time(const local\_time<Duration>& tp, const local\_info& i);
\end{verbatim}

\textbf{Preconditions:} \texttt{i.result == local\_info::ambiguous} is true.

\textbf{Effects:} Initializes the base class with a sequence of \texttt{char} equivalent to that produced by \texttt{os.str()} initialized as shown below:

\begin{verbatim}
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";
\end{verbatim}
4 Example:

```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() << '
';
    }
}
```

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

```cpp
namespace std::chrono {
    struct sys_info {
        sys_seconds begin;
        sys_seconds end;
        seconds offset;
        minutes save;
        string abbrev;
    };
}
```

1 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`.

2 [Note: 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]

3 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`.

4 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:

   ```cpp
   offset = local_time - sys_time
   ```

5 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`).

6 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.

```
template<class charT, class traits>
    basic_ostream<charT, traits>&
    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`. 

§ 27.11.4.1
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;
    };
}

[Note: 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]

Describes the result of converting a local_time to a sys_time as follows:

(2.1) — 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.

(2.2) — 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.

(2.3) — 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.

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const local_info& r);

Effects: Streams out the local_info object r in an unspecified format.

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: 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;

Returns: The name of the time_zone.

[Example: "America/New_York". — end example]

template<class Duration>
sys_info get_info(const sys_time<Duration>& st) const;

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;

Returns: A local_info for tp.

template<class Duration>
sys_time<common_type_t<Duration, seconds>>
to_sys(const local_time<Duration>& tp) const;

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.

Throws: If the conversion from tp to a sys_time is ambiguous, throws ambiguous_local_time. If the tp represents a non-existent time between two UTC time_points, throws nonexistent_local_time.

template<class Duration>
sys_time<common_type_t<Duration, seconds>>
to_sys(const local_time<Duration>& tp, choose z) const;

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;

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;

Returns: x.name() == y.name().

strong_ordering operator<=>(const time_zone& x, const time_zone& y) noexcept;

Returns: x.name() <=> y.name().

27.11.6 Class template zoned_traits

namespace std::chrono {
  template<class T> struct zoned_traits {};
}

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);
};

static const time_zone* default_zone();

Returns: \texttt{std::chrono::locate\_zone("UTC")}.

static const time_zone* locate_zone(string_view name);

Returns: \texttt{std::chrono::locate\_zone(name)}.

\section*{27.11.7 \textbf{Class template \texttt{zoned\_time}} [time.zone.zonedtime]}

\subsection*{27.11.7.1 \textbf{Overview} [time.zone.zonedtime.overview]}

\begin{verbatim}
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);

        template<class Duration2, class TimeZonePtr2>
        zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& zt);
        template<class Duration2, class TimeZonePtr2>
        zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& zt, choose);

        template<class Duration2, class TimeZonePtr2>
        zoned_time(string_view name, const zoned_time<Duration2, TimeZonePtr2>& zt);
        template<class Duration2, class TimeZonePtr2>
        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;
    };
}
\end{verbatim}
```cpp
sys_info get_info() const;

zoned_time() -> zoned_time<seconds>;

template<class Duration>
zoned_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>
zoned_time(TimeZonePtrOrName&&)
-> zoned_time<seconds, time-zone-representation<TimeZonePtrOrName>>;

template<class TimeZonePtrOrName, class Duration>
zoned_time(TimeZonePtrOrName&&, sys_time<Duration>)
-> zoned_time<common_type_t<Duration, seconds>,
    time-zone-representation<TimeZonePtrOrName>>;

template<class TimeZonePtrOrName, class Duration>
zoned_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>
zoned_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<Duration>` 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 `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 (12.4.1.8).

#### 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{})`.

§ 27.11.7.2
Effects: Initializes zone_ with traits::locate_zone(name) and default constructs tp_.

```cpp
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_.

```cpp
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.

```cpp
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}.

```cpp
zoned_time(TimeZonePtr z, const local_time<Duration>& tp);
```

Preconditions: z refers to a time zone.

```cpp
Constraints:
  is_convertible_v<
    decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>())),
    sys_time<Duration>>
  is true.
```

Effects: Initializes zone_ with std::move(z) and tp_ with zone_->to_sys(tp).

```cpp
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}.

```cpp
zoned_time(TimeZonePtr z, const local_time<Duration>&& tp, choose c);
```

Preconditions: z refers to a time zone.

```cpp
Constraints:
  is_convertible_v<
    decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>(), choose::earliest)),
    sys_time<Duration>>
  is true.
```

Effects: Initializes zone_ with std::move(z) and tp_ with zone_->to_sys(tp, c).

```cpp
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}.

```cpp
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: Initializes zone_ with std::move(z) and tp_ with y.tp_.

```cpp
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\}.

[Note: 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: \textit{zoned\_time} is constructible from the return type of \textit{traits::locate\_zone(name)} and the type \textit{zoned\_time<Duration2, TimeZonePtr2>}.  

Effects: Equivalent to construction with \{\textit{traits::locate\_zone(name)}, y\}.

template<class Duration2, class TimeZonePtr2>
  zoned_time(string_view name, const zoned_time<Duration2, TimeZonePtr2>& y, choose c);

Constraints: \textit{zoned\_time} is constructible from the return type of \textit{traits::locate\_zone(name)}, the type \textit{zoned\_time<Duration2, TimeZonePtr2>}, and the type \textit{choose}.

Effects: Equivalent to construction with \{\textit{traits::locate\_zone(name)}, y, c\}.

[Note: The \textit{choose} parameter has no effect. — end note]

27.11.7.3 Member functions

\texttt{zoned\_time\& operator\=(const sys\_time<Duration>& st);}  
1 Effects: After assignment, \textit{get\_sys\_time()} == \textit{st}. This assignment has no effect on the return value of \textit{get\_time\_zone()}.  
2 Returns: *this.

\texttt{zoned\_time\& operator\=(const local\_time<Duration>& lt);}  
3 Effects: After assignment, \textit{get\_local\_time()} == \textit{lt}. This assignment has no effect on the return value of \textit{get\_time\_zone()}.  
4 Returns: *this.

\texttt{operator sys\_time<duration>() const;}  
5 Returns: \textit{get\_sys\_time()}.  

\texttt{explicit operator local\_time<duration>() const;}  
6 Returns: \textit{get\_local\_time()}.  

TimeZonePtr get_time_zone() const;  
7 Returns: \textit{zone\_}.  

local\_time<duration> get_local\_time() const;  
8 Returns: \textit{zone\_}\rightarrow\textit{to\_local(tp\_)}.

sys\_time<duration> get_sys\_time() const;  
9 Returns: \textit{tp\_}.  

sys\_info get\_info() const;  
10 Returns: \textit{zone\_}\rightarrow\textit{get\_info(tp\_)}.

27.11.7.4 Non-member functions

\texttt{template<class Duration1, class Duration2, class TimeZonePtr>
  bool operator==\(\)(const zoned\_time<Duration1, TimeZonePtr>& x,  
  const zoned\_time<Duration2, TimeZonePtr>& y);}  
1 Returns: \textit{x}.\textit{zone\_} == \textit{y}.\textit{zone\_} \&\& \textit{x}.\textit{tp\_} == \textit{y}.\textit{tp\_}.

\texttt{template<class charT, class traits, class Duration, class TimeZonePtr>
  basic\_ostream<\texttt{charT}\texttt{, traits}>&
  operator\<(basic\_ostream<\texttt{charT}\texttt{, traits}>& os,  
  const zoned\_time<Duration, TimeZonePtr>& t);}  
2 Effects: Streams the value returned from \texttt{t.get\_local\_time()} to \texttt{os} using the format \texttt{"%F \%T \%Z"}.  

§ 27.11.7.4 1303
Returns: \texttt{os}.

27.11.8 Class \texttt{leap_second} \hfill \[time.zone.leap\]

27.11.8.1 Overview \hfill \[time.zone.leap.overview\]

namespace std::chrono {
    class leap_second {
    public:
        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 \texttt{leap_second} representing the date and value of the leap second insertions are constructed and stored in the time zone database when initialized.

[Example:

```cpp
for (auto& l : get_tzdb().leap_seconds)
    if (l <= 2018y/March/17d)
        cout << l.date() << " : " << l.value() << '\n';
```

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
1994-07-01 00:00:00: 1s
1996-01-01 00:00:00: 1s
1997-07-01 00:00:00: 1s
1999-01-01 00:00:00: 1s
2000-01-01 00:00:00: 1s
2001-01-01 00:00:00: 1s
2006-01-01 00:00:00: 1s
2009-01-01 00:00:00: 1s
2012-07-01 00:00:00: 1s
2015-07-01 00:00:00: 1s
2017-01-01 00:00:00: 1s

— end example]

27.11.8.2 Member functions \hfill \[time.zone.leap.members\]

    constexpr sys_seconds date() const noexcept;
1

    Returns: The date and time at which the leap second was inserted.

    constexpr seconds value() const noexcept;
2

    Returns: +1s to indicate a positive leap second or -1s to indicate a negative leap second. [Note: 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;
1
Returns: x.date() == y.date().

constexpr strong_ordering operator<=>(const leap_second& x, const leap_second& y) noexcept;
2
Returns: x.date() <=> y.date().

template<class Duration>
constexpr bool operator==(const leap_second& x, const sys_time<Duration>& y) noexcept;
3
Returns: x.date() == y.

template<class Duration>
constexpr bool operator<(const leap_second& x, const sys_time<Duration>& y) noexcept;
4
Returns: x.date() < y.

template<class Duration>
constexpr bool operator<(const sys_time<Duration>& x, const leap_second& y) noexcept;
5
Returns: x < y.date().

template<class Duration>
constexpr bool operator>(const leap_second& x, const sys_time<Duration>& y) noexcept;
6
Returns: y < x.

template<class Duration>
constexpr bool operator>(const sys_time<Duration>& x, const leap_second& y) noexcept;
7
Returns: y < x.

template<class Duration>
constexpr bool operator<=(const leap_second& x, const sys_time<Duration>& y) noexcept;
8
Returns: !(y < x).

template<class Duration>
constexpr bool operator<=(const sys_time<Duration>& x, const leap_second& y) noexcept;
9
Returns: !(y < x).

template<class Duration>
constexpr bool operator>=(const leap_second& x, const sys_time<Duration>& y) noexcept;
10
Returns: !(x < y).

template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap_second& y) noexcept;
11
Returns: !(x < y).

requires three_way_comparable_with<sys_seconds, sys_time<Duration>>
constexpr auto operator<=>(const leap_second& x, const sys_time<Duration>& y) noexcept;
12
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
    }
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

```cpp
class time_zone_link {
    string_view name() const noexcept;
    string_view target() const noexcept;
};
```

1. `string_view name()`
   Returns: The alternative name for the time zone.

2. `string_view target()`
   Returns: The name of the `time_zone` for which this `time_zone_link` provides an alternative name.

### 27.11.9.3 Non-member functions

```cpp
bool operator==(const time_zone_link& x, const time_zone_link& y) noexcept;
strong_ordering operator<=>(const time_zone_link& x, const time_zone_link& y) noexcept;
```

1. `bool operator==(const time_zone_link& x, const time_zone_link& y) noexcept`
   Returns: `x.name() == y.name()`.

2. `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:

```plaintext
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 O
    type: one of a A b B c C d D e E f F g G h H i I j m M n p q r s t T u u v w W x X y Y z z %
```

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. 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:

```cpp
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}", -1'000s);
   // prints: minutes -46, hours 02, seconds 40

— end example]}

4 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: If the information is not available and a %Z or %z conversion specifier appears in the chronoformat-spec, an exception of type format_error is thrown, as described above. — end note]

5 If the type being formatted does not contain the information that the format flag needs, an exception of type format_error is thrown. [Example: 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>%a</td>
<td>The locale’s abbreviated weekday name. If the value does not contain a valid weekday, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%A</td>
<td>The locale’s full weekday name. If the value does not contain a valid weekday, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%b</td>
<td>The locale’s abbreviated month name. If the value does not contain a valid month, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%B</td>
<td>The locale’s full month name. If the value does not contain a valid month, an exception of type format_error is thrown.</td>
</tr>
<tr>
<td>%c</td>
<td>The locale’s date and time representation. The modified command %Ec produces the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td>%C</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 %EC produces the locale’s alternative representation of the century.</td>
</tr>
<tr>
<td>%d</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 %Od produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%D</td>
<td>Equivalent to %m/%d/%y.</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 %Oe 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 %OH 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 %OI produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%j</td>
<td>If the type being formatted is a specialization of duration, 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 %Om 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 %OM 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>
</tbody>
</table>
### Table 99: Meaning of conversion specifiers (continued)

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>%q</td>
<td>The duration’s unit suffix as specified in 27.5.10.</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 microsecond 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 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. 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>
<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 %0y produces the locale’s alternative representation. The modified command %Ey produces the locale’s alternative representation of offset from %EC (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 %0Y produces the locale’s alternative full year representation.</td>
</tr>
<tr>
<td>%z</td>
<td>The offset from UTC in the ISO 8601 format. For example -04:30 refers to 4 hours 30 minutes behind UTC. If the offset is zero, +0000 is used. The modified commands %0z and %0z insert a : between the hours and minutes: -04:30. If the offset information is not available, an exception of type <code>format_error</code> 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 <code>format_error</code> 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::ostringstream` `os` and copying `os.str()` through the output iterator of the context with additional padding and adjustments as specified by the format specifiers. [Example:]

```cpp
    string s = format("{:==>8}", 42ms);  // value of s is "====42ms"
    // end example]```
template<class Duration, class charT>
struct formatter<chrono::sys_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("UTC"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted.

template<class Duration, class charT>
struct formatter<chrono::utc_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("UTC"). If %z (or a modified variant of %z) is used, an offset of 0min is formatted. If the argument represents a time during a positive leap second insertion, and if a seconds field is formatted, the integral portion of that format is STATICALLY-WIDEN<charT>("60").

template<class Duration, class charT>
struct formatter<chrono::tai_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("TAI"). 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 sys_time initialized with

sys_time<Duration>({tp.time_since_epoch()} - (sys_days{1970y/January/1} - sys_days{1958y/January/1}))

template<class Duration, class charT>
struct formatter<chrono::gps_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<charT>("GPS"). 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 sys_time initialized with

sys_time<Duration>({tp.time_since_epoch()} + (sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1}))

template<class Duration, class charT>
struct formatter<chrono::file_time<Duration>, charT>;

Remarks: If %Z is used, it is replaced with STATICALLY-WIDEN<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 sys_time initialized with clock_cast<system_clock>(t), or by a utc_time initialized with clock_cast<utc_clock>(t), where t is the first argument to 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 format_error is thrown.

template<class Duration> struct local_time-format-t { // exposition only
  local_time<Duration> time;
  const string* abbrev;
  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);

Returns: {time, abbrev, offset_sec}.

template<class Duration, class charT>
struct formatter<chrono::local_time-format-t<Duration>, charT>;

Let f be a local_time-format-t<Duration> object passed to 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 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 format_error is thrown.
template<class Duration, class TimeZonePtr, class charT>
struct formatter<chrono::zoned_time<Duration, TimeZonePtr>, charT> :
formatter<chrono::local_time_format_t<Duration>, charT> {
    template<class FormatContext>
    typename FormatContext::iterator
    format(const chrono::zoned_time<Duration, TimeZonePtr>& tp, FormatContext& ctx);
};

template<class FormatContext>
typename FormatContext::iterator
format(const chrono::zoned_time<Duration, TimeZonePtr>& tp, FormatContext& ctx);

Effects: Equivalent to:
        sys_info info = tp.get_info();
        return formatter<chrono::local_time_format_t<Duration>, charT>::
        format({tp.get_local_time(), &info.abbrev, &info.offset}, ctx);

27.13 Parsing

Each parse overload specified in this subclause calls from_stream unqualified, so as to enable argument
dependent lookup (6.5.2). In the following paragraphs, let is denote an object of type basic_istream<charT,
traits> and let I be basic_istream<charT, traits>&, where charT and traits are template parameters
in that context.

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp);

Constraints: The expression
                from_stream(declval<basic_istream<charT, traits>>&()), fmt.c_str(), tp)
            is well-formed when treated as an unevaluated operand.

Returns: A manipulator such that the expression is >> parse(fmt, tp) has type I, has value is,
            and calls from_stream(is, fmt.c_str(), tp).

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
        basic_string<charT, traits, Alloc>& abbrev);

Constraints: The expression
                from_stream(declval<basic_istream<charT, traits>>&()), fmt.c_str(), tp,
                addressof(abbrev))
            is well-formed when treated as an unevaluated operand.

Returns: A manipulator such that the expression is >> parse(fmt, tp, abbrev) has type I, has
            value is, and calls from_stream(is, fmt.c_str(), tp, addressof(abbrev)).

template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
        minutes& offset);

Constraints: The expression
                from_stream(declval<basic_istream<charT, traits>>&()),
                fmt.c_str(), tp,
                declval<basic_string<charT, traits, Alloc>*>(),
                &offset)
            is well-formed when treated as an unevaluated operand.

Returns: A manipulator such that the expression is >> parse(fmt, tp, offset) has type I, has
            value is, and calls:
                from_stream(is,
                fmt.c_str(), tp,
                static_cast<basic_string<charT, traits, Alloc>*>(nullptr),
                &offset)
template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
       basic_string<charT, traits, Alloc>& abbrev, minutes& offset);

8 Constraints: The expression
   from_stream(declval<basic_istream<charT, traits>&>(),
                fmt.c_str(), tp, addressof(abbrev), &offset)

is well-formed when treated as an unevaluated operand.

9 Returns: A manipulator such that the expression
   is >> parse(fmt, tp, abbrev, offset) has type
I, has value is, and calls from_stream(is, fmt.c_str(), tp, addressof(abbrev), &offset).

10 All from_stream overloads behave as unformatted input functions, except that they have an unspecified
   effect on the value returned by subsequent calls to basic_istream::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
white space, are parsed unchanged from the stream. A white space character matches zero or more white
space characters in the input stream.

11 If the type being parsed cannot represent the information that the format flag refers to, is.setstate(ios_base::failbit) is called. [Example: 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 duration is parsed as the
time of day elapsed since midnight.

12 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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>The locale’s full or abbreviated case-insensitive weekday name.</td>
</tr>
<tr>
<td>%A</td>
<td>Equivalent to %a.</td>
</tr>
<tr>
<td>%b</td>
<td>The locale’s full or abbreviated case-insensitive month name.</td>
</tr>
<tr>
<td>%B</td>
<td>Equivalent to %b.</td>
</tr>
<tr>
<td>%c</td>
<td>The locale’s date and time representation. The modified command %Ec interprets the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td>%C</td>
<td>The century as a decimal number. The modified command ( %NC ) specifies the maximum number of characters to read. If ( N ) is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %EC interprets the locale’s alternative representation of the century.</td>
</tr>
<tr>
<td>%d</td>
<td>The day of the month as a decimal number. The modified command ( %Nd ) specifies the maximum number of characters to read. If ( N ) is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %Od interprets the locale’s alternative representation of the day of the month.</td>
</tr>
<tr>
<td>%D</td>
<td>Equivalent to %m/%d/%y.</td>
</tr>
<tr>
<td>%e</td>
<td>Equivalent to %d and can be modified like %d.</td>
</tr>
<tr>
<td>%f</td>
<td>Equivalent to %y-%m-%d. If modified with a width ( N ), the width is applied to only %y.</td>
</tr>
<tr>
<td>%g</td>
<td>The last two decimal digits of the ISO week-based year. The modified command %Ng specifies the maximum number of characters to read. If ( N ) is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%G</td>
<td>The ISO week-based year as a decimal number. The modified command %NG specifies the maximum number of characters to read. If ( N ) is not specified, the default is 4. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td>%h</td>
<td>Equivalent to %b.</td>
</tr>
</tbody>
</table>
Table 100: Meaning of parse flags (continued)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%H</code></td>
<td>The hour (24-hour clock) as a decimal number. The modified command %\mH\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0H\n\ 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 %\mI\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0I\n\ interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%j</code></td>
<td>If the type being parsed is a specialization of duration, a decimal number of days. Otherwise, the day of the year as a decimal number. Jan 1 is 1. In either case, the modified command %\mj\n\ specifies the maximum number of characters to read. If N 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 %\mm\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0m\n\ interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%M</code></td>
<td>The minutes as a decimal number. The modified command %\mM\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0m\n\ interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%n</code></td>
<td>Matches one white space character. [Note: %n, %t, and a space can be combined to match a wide range of white-space patterns. For example, &quot;%n\n&quot; matches one or more white space characters, and &quot;%n%t%t&quot; matches one to three white space characters. — end note]</td>
</tr>
<tr>
<td><code>%p</code></td>
<td>The locale’s equivalent of the AM/PM designations associated with a 12-hour clock.</td>
</tr>
<tr>
<td><code>%r</code></td>
<td>The locale’s 12-hour clock time.</td>
</tr>
<tr>
<td><code>%S</code></td>
<td>The seconds as a decimal number. The modified command %\mS\n\ specifies the maximum number of characters to read. If N 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 long double in a fixed format. If encountered, the locale determines the decimal point character. Leading zeroes are permitted but not required. The modified command %0S\n\ interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%t</code></td>
<td>Matches zero or one white space characters.</td>
</tr>
<tr>
<td><code>%T</code></td>
<td>Equivalent to %\mH:\mM:\mS.</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 %\mu\n\ specifies the maximum number of characters to read. If N 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 %\mu\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0U\n\ interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td><code>%w</code></td>
<td>The ISO week-based week number as a decimal number. The modified command %\mw\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required.</td>
</tr>
<tr>
<td><code>%W</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 %\mW\n\ specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0W\n\ interprets the locale’s alternative representation.</td>
</tr>
</tbody>
</table>

§ 27.13 1312
### Table 100: Meaning of parse flags (continued)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Parsed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%W</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 %NW specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified command %0W interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%x</td>
<td>The locale’s date representation. The modified command %Ex interprets the locale’s alternate date representation.</td>
</tr>
<tr>
<td>%X</td>
<td>The locale’s time representation. The modified command %EX interprets the locale’s alternate time representation.</td>
</tr>
<tr>
<td>%y</td>
<td>The last two decimal digits of the year. If the century is not otherwise specified (e.g. with %C), 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 %Ny specifies the maximum number of characters to read. If N is not specified, the default is 2. Leading zeroes are permitted but not required. The modified commands %Ey and %Oy interpret the locale’s alternative representation.</td>
</tr>
<tr>
<td>%Y</td>
<td>The year as a decimal number. The modified command %NY specifies the maximum number of characters to read. If N is not specified, the default is 4. Leading zeroes are permitted but not required. The modified command %EY interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%z</td>
<td>The offset from UTC in the format [+</td>
</tr>
<tr>
<td>%Z</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>%%</td>
<td>A % character is extracted.</td>
</tr>
</tbody>
</table>

#### 27.14 Header `<ctime>` synopsis

```cpp
#define NULL see 17.2.3
#define CLOCKS_PER_SEC see below
#define TIME_UTC see below

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);
}
```

The contents of the header `<ctime>` are the same as the C standard library header `<time.h>`.

---

255) `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 O.
The functions `asctime`, `ctime`, `gmtime`, and `localtime` are not required to avoid data races (16.5.5.10). See also: ISO C 7.27
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.

28.2 Header <locale> synopsis

```cpp
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);

// 28.4.1, ctype
class ctype_base;
template<class charT> class ctype;
template<> class ctype<char>;
// specialization
template<class charT> class ctype_byname;
class codecvt_base;
template<class internT, class externT, class stateT> class codecvt;
template<class internT, class externT, class stateT> class codecvt_byname;

// 28.4.2, 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;

// 28.4.4, collation
template<class charT>
class collate;
template<class charT>
class collate_byname;

// 28.4.5, 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;

// 28.4.6, 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;

// 28.4.7, message retrieval
class messages_base;
template<class charT>
class messages;
template<class charT>
class messages_byname;
}

The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.

28.3 Locales

28.3.1 Class locale

namespace std
{
class locale {
public:
    // types
    class facet;
    class id;
    using category = int;
    static const category 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);
    locale(const locale& other, const string& std_name, category);
template<class Facet> locale(const locale& other, Facet* f);

256) In this subclause, the type name `struct tm` is an incomplete type that is defined in `<ctime>` (27.14).
locale(const locale& other, const locale& one, category);
~locale(); // not virtual
const locale& operator=(const locale& other) noexcept;
template<class Facet> 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: An iostream operator<< might be implemented as:]

```
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&& s, Date d) {
    typename basic_ostream<charT, traits>::sentry cerberos(s);
    if (cerberos) {
        tm tmbuf; d.extract(tmbuf);
        bool failed =
            use_facet<time_put<charT, ostreambuf_iterator<charT, traits>>>(s.getloc()).put(s, s, s.fill(), &tmbuf, 'x').failed();
        if (failed)
            s.setstate(s.badbit); // might throw
    }
    return s;
}
```

—end example]

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: All locale semantics are accessed via use_facet<> and has_facet<>], except that:

(5.1) — A member operator template operator()(const basic_string<C, T, A>&, const basic_string<C, T, A>&) is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

(5.2) — Convenient global interfaces are provided for traditional cctype functions such as isdigit() and isspace(), so that given a locale object loc a C++ program can call isspace(c, loc). (This eases upgrading existing extractors (29.7.4.2).)

—end note]

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.

257) Note that in the call to put, the stream is implicitly converted to an ostreambuf_iterator<charT, traits>.

§ 28.3.1 1317
In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

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 "*".

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 (16.5.5.10).

### 28.3.1.1 Types

**28.3.1.1.1 Type `locale::category`**

```c++
using category = int;
```

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.

`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>(l)` 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  

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td><code>collate_byname&lt;char&gt;</code>, <code>collate_byname&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td>ctype</td>
<td><code>ctype_byname&lt;char&gt;</code>, <code>ctype_byname&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt_byname&lt;char, char, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt_byname&lt;char16_t, char8_t, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt_byname&lt;char32_t, char8_t, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt_byname&lt;wchar_t, char, mbstate_t&gt;</code></td>
</tr>
<tr>
<td>monetary</td>
<td><code>moneypunct_byname&lt;char, International&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>moneypunct_byname&lt;wchar_t, International&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>money_get&lt;C, InputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>money_put&lt;C, OutputIterator&gt;</code></td>
</tr>
<tr>
<td>numeric</td>
<td><code>numpunct_byname&lt;char&gt;</code>, <code>numpunct_byname&lt;wchar_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>num_get&lt;C, InputIterator&gt;</code>, <code>num_put&lt;C, OutputIterator&gt;</code></td>
</tr>
<tr>
<td>time</td>
<td><code>time_get&lt;char, InputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_get_byname&lt;char, InputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_get&lt;wchar_t, InputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_get_byname&lt;wchar_t, InputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_put&lt;char, OutputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_put_byname&lt;char, OutputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_put&lt;wchar_t, OutputIterator&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>time_put_byname&lt;wchar_t, OutputIterator&gt;</code></td>
</tr>
<tr>
<td>messages</td>
<td><code>messages_byname&lt;char&gt;</code>, <code>messages_byname&lt;wchar_t&gt;</code></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.

28.3.1.1.2 Class `locale::facet`  

```cpp
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:

```cpp
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.

---

258) 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.
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.

28.3.1.1.3 Class `locale::id` [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.

[Note: 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]

28.3.1.2 Constructors and destructor [locale.cons]

```cpp
locale() noexcept;
```

1 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: 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);
```

2 Effects: Constructs a locale using standard C locale names, e.g., "POSIX". The resulting locale implements semantics defined to be associated with that name.

3 Throws: `runtime_error` if the argument is not valid, or is null.

4 Remarks: The set of valid string argument values is "C", "", and any implementation-defined values.

```cpp
explicit locale(const string& std_name);
```

5 Effects: The same as `locale(std_name.c_str())`.

```cpp
locale(const locale& other, const char* std_name, category);
```

6 Effects: Constructs a locale as a copy of `other` except for the facets identified by the `category` argument, which instead implement the same semantics as `locale(std_name)`.

7 Throws: `runtime_error` if the argument is not valid, or is null.

8 Remarks: The locale has a name if and only if `other` has a name.

```cpp
locale(const locale& other, const string& std_name, category cat);
```

9 Effects: The same as `locale(other, std_name.c_str(), cat)`.

```cpp
template<class Facet> locale(const locale& other, Facet* f);
```

10 Effects: Constructs a locale incorporating all facets from the first argument except that of type `Facet`, and installs the second argument as the remaining facet. If `f` is null, the resulting object is a copy of `other`.

11 Remarks: The resulting locale has no 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.

28.3.1.3 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 "*".

28.3.1.4 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.

Remarks: This member operator template (and therefore locale itself) meets the requirements for a comparator predicate template argument (Clause 25) applied to strings.

Returns:
use_facet<collate<charT>>(*this).compare(s1.data(), s1.data() + s1.size(), s2.data(), s2.data() + s2.size()) < 0

[Example: A vector of strings v can be collated according to collation rules in locale loc simply by (25.8.1, 22.3.11):
std::sort(v.begin(), v.end(), loc);
—end example]

28.3.1.5 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.

Remarks: No library function other than locale::global() affects the value returned by locale().
[Note: See 28.5 for data race considerations when setlocale is invoked. — end note]

Returns: The previous value of locale().

§ 28.3.1.5
static const locale& classic();

The "C" locale.

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

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 28.3.1.1.2.

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.

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

Each of these functions isF returns the result of the expression:

use_facet<ctype<charT>>(loc).is(ctype_base::F, c)

where F is the ctype_base::mask value corresponding to that function (28.4.1).

28.3.3.2 Conversions

28.3.3.2.1 Character conversions

Each such member function takes an 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 put() members make no provision for error reporting. (Any failures of the OutputIterator argument can be extracted from the returned iterator.) The get() members take an ios_base::iostate& argument whose value they ignore, but set to ios_base::failbit in case of a parse error.

259) When used in a loop, it is faster to cache the ctype<> facet and use it directly, or use the vector form of ctype<>::is.
Within this clause it is unspecified whether one virtual function calls another virtual function.

28.4.1 The ctype category

namespace std {
    class ctype_base {
        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;
    }
}

The type mask is a bitmask type (16.4.2.2.4).

28.4.1.1 Class template ctype

namespace std {
    template<class charT>
    class ctype : public locale::facet, public ctype_base {
        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;
    }
    ~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_tolower(charT) const;
    virtual const charT* do_tolower(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;
}

§ 28.4.1.1 1323
virtual const charT* do_narrow(const charT* low, const charT* high,
    char dfault, char* 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 (28.3.1.1.1), namely `ctype<char>` and `ctype<wchar_t>`, implement character classing appropriate to the implementation’s native character set.

### 28.4.1.1.1 `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 c) const;**
- **const charT* toupper(charT* low, const charT* high) const;**

4 **Returns:** `do_toupper(c) or do_toupper(low, high).`

- **tolower(charT c) const;**
- **const charT* tolower(charT* low, const charT* high) const;**

5 **Returns:** `do_tolower(c) or do_tolower(low, high).`

- **charT widen(char c) const;**
- **const char* widen(const char* low, const char* high, charT* to) const;**

6 **Returns:** `do_widen(c) or do_widen(low, high, to).`

- **charT 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).`

### 28.4.1.1.2 `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`.  

§ 28.4.1.1.2 1324
charT do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;

7 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.

ccharT do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;

9 Effects: Converts a character or characters to lower case. The second form replaces each character *p in the range [low, high] 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.

ccharT do_widen(char c) const;
const char* do_widen(const char* low, const char* high, charT* dest) const;

11 Effects: Applies the simplest reasonable transformation from a char value or sequence of char values to the corresponding charT value or values.\(^{260}\) 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.\(^{261}\) 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.

cchar do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;

13 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.

28.4.1.2 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();

260) The char argument of do_widen is intended to accept values derived from character-literals for conversion to the locale’s encoding.
261) In other words, the transformed character is not a member of any character classification that c is not also a member of.
28.4.1.3  ctype<char> specialization

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;

        static locale::id id;
        static const size_t table_size = implementation-defined;
        const mask* table() const noexcept;
        static const mask* classic_table() noexcept;
    };
}

A specialization ctype<char> is provided so that the member functions on type char can be implemented inline.\textsuperscript{262} The implementation-defined value of member table_size is at least 256.

28.4.1.3.1  Destructor

~ctype();

Effects: If the constructor’s first argument was nonzero, and its second argument was true, does delete [] table().

28.4.1.3.2  Members

In the following member descriptions, for unsigned char values \( v \) where \( v \geq \) table_size, \( table()[v] \) is assumed to have an implementation-specific value (possibly different for each such value \( v \)) without performing the array lookup.

\textsuperscript{262} 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>.
explicit ctype(const mask* tbl = nullptr, bool del = false, size_t refs = 0);

Preconditions: Either tbl == nullptr is true or [tbl, tbl+table_size) is a valid range.

Effects: Passes its refs argument to its base class constructor.

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 vec[p - low] the value

Returns: The first form returns table()[(unsigned char)c] & m; the second form returns high.

const char* scan_is(mask m, const char* low, const char* high) const;

Returns: The smallest p in the range [low, high) such that

is true.

const char* scan_not(mask m, const char* low, const char* high) const;

Returns: The smallest p in the range [low, high) such that

is false.

char toupper(char c) const;
const char* toupper(char* low, const char* high) const;

Returns: do_toupper(c) or do_toupper(low, high), respectively.

char tolower(char c) const;
const char* tolower(char* low, const char* high) const;

Returns: do_tolower(c) or do_tolower(low, high), respectively.

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.

char narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high, char dfault, char* to) const;

Returns: do_narrow(c, dfault) or do_narrow(low, high, dfault, to), respectively.

const mask* table() const noexcept;

Returns: The first constructor argument, if it was nonzero, otherwise classic_table().

28.4.1.3.3 Static members

static const mask* classic_table() noexcept;

Returns: A pointer to the initial element of an array of size table_size which represents the classifica-
tions of characters in the "C" locale.

28.4.1.3.4 Virtual functions

These functions are described identically as those members of the same name in the ctype class template
(28.4.1.1.1).
28.4.1.4 Class template codecvt

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&, 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&, const externT* from, const externT* end, size_t max) const;
        virtual int do_max_length() const noexcept;
    }
}

1 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.

2 The stateT argument selects the pair of character encodings being mapped between.
The specializations required in Table 102 (28.3.1.1.1) 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 program-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.

### 28.4.1.4.1 Members

#### [locale.codecvt.members]

- **result out**
  
  ```
  stateT& state,
  const internT* from, const internT* from_end, const internT*& from_next,
  externT* to, externT* to_end, externT*& to_next) const;
  ```

  **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;
  ```

  **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;
  ```

  **Returns:**
  
  ```
  do_in(state, from, from_end, from_next, to, to_end, to_next);
  ```

- **int encoding() const noexcept;**
  
  **Returns:**
  
  ```
  do_encoding();
  ```

- **bool always_noconv() const noexcept;**
  
  **Returns:**
  
  ```
  do_always_noconv();
  ```

- **int length(stateT& state, const externT* from, const externT* from_end, size_t max) const;**
  
  **Returns:**
  
  ```
  do_length(state, from, from_end, max);
  ```

### 28.4.1.4.2 Virtual functions

#### [locale.codecvt.virtuals]

- **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;
  ```

- **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;
  ```

  **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.

  **Effects:**
  
  Translates characters in the source range `[from, from_end)`, placing the results in sequential positions starting at destination `to`. Converts no more than `(from_end - from)` source elements, and stores no more than `(to_end - to)` destination elements.

  Stops if it encounters a character it cannot convert. It always leaves the `from_next` and `to_next` pointers pointing one beyond the last element successfully converted. If returns `noconv`, `internT` and `externT` are the same type and the converted sequence is identical to the input sequence `[from, from_next)`. 

§ 28.4.1.4.2
to_next is set equal to to, the value of state is unchanged, and there are no changes to the values in [to, to_end).

A codecvt facet that is used by basic_filebuf (29.9) shall have the property that if
do_out(state, from, from_end, from_next, to, to_end, to_next)
would return ok, where from != from_end, then
do_out(state, from, from + 1, from_next, to, to_end, to_next)
shall also return ok, and that if
do_in(state, from, from_end, from_next, to, to_end, to_next)
would return ok, where to != to_end, then
do_in(state, from, from_end, from_next, to, to + 1, to_next)
shall also return ok.263 [Note: As a result of operations on state, it can return ok or partial and set from_next == from_end == to. — end note]

Remarks: Its operations on state are unspecified. [Note: 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. — 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 [from, from_end)</td>
</tr>
<tr>
<td></td>
<td>that it could not convert</td>
</tr>
<tr>
<td>noconv</td>
<td>internT and externT are the same type, and input sequence is identical to converted sequence</td>
</tr>
</tbody>
</table>

A return value of partial, if (from_next == from_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 do_unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;

Preconditions: (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.

Effects: Places characters starting at to that should be appended to terminate a sequence when the current stateT is given by state.264 Stores no more than (to_end - to) destination elements, and leaves the to_next pointer pointing one beyond the last element successfully stored.

Returns: An enumeration value, as summarized in Table 105.

<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>

263 Informally, this means that basic_filebuf assumes that the mappings from internal to external characters is 1 to N: a codecvt facet that is used by basic_filebuf must be able to translate characters one internal character at a time.

264 Typically these will be characters to return the state to stateT().
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.

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.

28.4.1.5 Class template 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);

    protected:
        ~codecvtbyname();
    };
}

28.4.2 The numeric category

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.

All specifications of member functions for num_put and num_get in the subclauses of 28.4.2 only apply to the specializations required in Tables 102 and 103 (28.3.1.1.1), namely num_get<char>, num_get<wchar_t>, num_get<C, InputIterator>, num_put<char>, num_put<wchar_t>, and num_put<C, OutputIterator>. These specializations refer to the ios_base::argument for formatting specifications (28.4), and to its imbued locale for the mumpunct<> facet to identify all numeric punctuation preferences, and also for the ctype<> 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 (29.7.4.2.1, 29.7.5.2.1).

If encoding() yields -1, then more than max_length() externT elements may be consumed when producing a single internT character, and additional externT elements may appear at the end of a sequence after those that yield the final internT character.

Parsing “-1” correctly into, e.g., an unsigned short requires that the corresponding member get() at least extract the sign before delegating.
28.4.2.1 Class template num_get

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, iter_type, ios_base&,
            ios_base::iostate& err, bool& v) const;
        virtual iter_type do_get(iter_type, iter_type, ios_base&,
            ios_base::iostate& err, long& v) const;
        virtual iter_type do_get(iter_type, iter_type, ios_base&,
            ios_base::iostate& err, long long& v) const;
        virtual iter_type do_get(iter_type, iter_type, ios_base&,
            ios_base::iostate& err, unsigned short& v) const;
        virtual iter_type do_get(iter_type, iter_type, ios_base&,
            ios_base::iostate& err, unsigned int& v) const;
        virtual iter_type do_get(iter_type, iter_type, ios_base&,
            ios_base::iostate& err, unsigned long& v) const;
        virtual iter_type do_get(iter_type, iter_type, 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;
    
    };

1 The facet num_get is used to parse numeric values from an input sequence such as an istream.
28.4.2.1.1 Members

```cpp
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)`.

28.4.2.1.2 Virtual functions

```cpp
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()`.

2 The details of this operation occur in three stages

(2.1) Stage 1: Determine a conversion specifier

(2.2) Stage 2: Extract characters from `in` and determine a corresponding `char` value for the format expected by the conversion specification determined in stage 1.

(2.3) Stage 3: Store results

3 The details of the stages are presented below.

**Stage 1:** The function initializes local variables via

```cpp
fmtflags flags = str.flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
```
fmtflags boolalpha = (flags & ios_base::boolalpha);

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>%o</td>
</tr>
<tr>
<td>basefield == hex</td>
<td>%X</td>
</tr>
<tr>
<td>basefield == 0</td>
<td>%i</td>
</tr>
<tr>
<td>signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

For conversions to a floating-point type the specifier is %g.

For conversions to void* the specifier is %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>h</td>
</tr>
<tr>
<td>unsigned short</td>
<td>h</td>
</tr>
<tr>
<td>long</td>
<td>l</td>
</tr>
<tr>
<td>unsigned long</td>
<td>l</td>
</tr>
<tr>
<td>long long</td>
<td>ll</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>ll</td>
</tr>
<tr>
<td>double</td>
<td>l</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
</tbody>
</table>

Stage 2: If in == end then stage 2 terminates. Otherwise a charT is taken from in and local variables are initialized as if by

```cpp
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;
```

where the values src and atoms are defined as if by:

```cpp
static const char src[] = "0123456789abcdefABCDEF+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT>>(loc).widen(src, src + sizeof(src), atoms);
```

for this value of loc.

If discard is 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 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 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 <cstdlib>:

1. For a signed integer value, the function `strtoll`
2. For an unsigned integer value, the function `strtoull`
3. For a `float` value, the function `strtof`
4. For a `double` value, the function `strtod`
5. For a long double value, the function `strtold`

§ 28.4.2.1.2
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, `ios_base::failbit` is assigned to `err`.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with `use_facet<numpunct<charT>>(loc).grouping()`. If they are not consistent then `ios_base::failbit` is assigned to `err`.

In any case, if stage 2 processing was terminated by the test for `in == end` then `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 `(str.flags()&ios_base::boolalpha) == 0` then input proceeds as it would for a `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 `false` is stored. If the value is 1 then `true` is stored. Otherwise `true` is stored and `ios_base::failbit` is assigned to `err`.

Otherwise target sequences are determined “as if” by calling the members `falsename()` and `truename()` of the facet obtained by `use_facet<numpunct<charT>>(str.getloc())`. Successive characters in the range `[in, end)` (see 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 `end` only when necessary to obtain a character. If a target sequence is uniquely matched, `val` is set to the corresponding value. Otherwise `false` is stored and `ios_base::failbit` is assigned to `err`.

The `in` iterator is always left pointing one position beyond the last character successfully matched. If `val` is set, then `err` is set to `str.goodbit`; or to `str.eofbit` if, when seeking another character to match, it is found that `(in == end)`. If `val` is not set, then `err` is set to `str.failbit`; or to `(str.failbit|str.eofbit)` if the reason for the failure was that `(in == end)`. [Example: For targets `true`: "a" and false: "abb", the input sequence "a" yields `val == true` and `err == str.eofbit`; the input sequence "abc" yields `err = str.failbit`, with `in` ending at the 'c' element. For targets `true`: "1" and false: "0", the input sequence "1" yields `val == true` and `err == str.goodbit`. For empty targets (""), any input sequence yields `err == str.failbit`. — end example]

Returns: `in`.

### 28.4.2.2 Class template `num_put`

```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, const void* v) const;

    static locale::id id;
}
```

§ 28.4.2.2
The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

**28.4.2.2.1 Members**

```cpp
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;
};
```

1 The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

**28.4.2.2.2 Virtual functions**

```cpp
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)`.

**28.4.2.2.2 Effects**

```cpp
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();
```

The details of this operation occur in several stages:

1. **Stage 1:** Determine a printf conversion specifier `spec` and determine the characters that would be printed by `printf` (29.12) given this conversion specifier for

```
printf(spec, val)
```

assuming that the current locale is the "C" locale.

2. **Stage 2:** Adjust the representation by converting each `char` determined by stage 1 to a `charT` using a conversion and values returned by members of `use_facet<numpunct<charT>>(loc)`.

3. **Stage 3:** Determine where padding is required.

4. **Stage 4:** Insert the sequence into the `out`.

1 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>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == ios_base::oct</td>
<td><code>%o</code></td>
</tr>
<tr>
<td>(basefield == ios_base::hex) &amp;&amp; !uppercase</td>
<td><code>%x</code></td>
</tr>
<tr>
<td>(basefield == ios_base::hex)</td>
<td><code>%X</code></td>
</tr>
<tr>
<td>for a signed integral type</td>
<td><code>%d</code></td>
</tr>
<tr>
<td>for an unsigned 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>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatfield == ios_base::fixed</td>
<td><code>%f</code></td>
</tr>
<tr>
<td>floatfield == ios_base::scientific &amp;&amp; !uppercase</td>
<td><code>%e</code></td>
</tr>
<tr>
<td>floatfield == ios_base::scientific</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>!uppercase</td>
<td><code>%g</code></td>
</tr>
<tr>
<td>otherwise</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>
<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>ll</td>
</tr>
<tr>
<td>unsigned long</td>
<td>1</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>ll</td>
</tr>
<tr>
<td>long double</td>
<td>L</td>
</tr>
<tr>
<td>otherwise</td>
<td>none</td>
</tr>
</tbody>
</table>

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 111.

<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 a is the conversion specifier determined above.

Stage 2: Any character c other than a decimal point(.) is converted to a charT via
use_facet<ctype<charT>>>(loc).widen(c)
A local variable punct is initialized via
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 28.4.3.1.2

Decimal point characters(;) are replaced by punct.decimal_point()

Stage 3: A local variable is initialized as
fmtflags adjustfield = (flags & (ios_base::adjustfield));
The location of any padding is determined according to Table 112.

Table 112: Fill padding

<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
*out++ = c
iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;

Returns: If (str.flags() & ios_base::boolalpha) == 0 returns do_put(out, str, fill, (int)val), otherwise obtains a string s as if by

```
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.

28.4.3 The numeric punctuation facet

28.4.3.1 Class template numpunct

```cpp
namespace std {
  template<class charT>
  class numpunct : public locale::facet {
    public:
      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;
  }
}
```

267) The conversion specification #o generates a leading 0 which is not a padding character.
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 (28.3.1.1.1), 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:
  sign_opt units
  sign:
  +
  -
  units:
  digits
digits thousands-sep units
digits:
  digit digits_opt
```

and floating-point values have:

```
floatval:
  sign_opt units fractional_opt exponent_opt
  sign_opt decimal-point digits exponent_opt
fractional:
  decimal-point digits_opt
exponent:
  e sign_opt 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.

28.4.3.1.1 Members [facet.numpunct.members]

```c
char_type decimal_point() const;
Returns: do_decimal_point().
```

```c
char_type thousands_sep() const;
Returns: do_thousands_sep().
```

```c
string grouping() const;
Returns: do_grouping().
```

```c
string_type truename() const;
string_type falsename() const;
Returns: do_truename() or do_falsename(), respectively.
```
28.4.3.1.2 Virtual functions

char_type do_decimal_point() const;

1 Returns: A character for use as the decimal radix separator. The required specializations return '.' or L'.'.

cchar_type do_thousands_sep() const;

2 Returns: A character for use as the digit group separator. The required specializations return ',' or L','.

string do_grouping() const;

3 Returns: A string vec used as a vector of integer values, in which each element vec[i] represents the number of digits \(^{268}\) in the group at position i, starting with position 0 as the rightmost group. If vec.size() <= i, the number is the same as group \((i - 1)\); if \((i < 0 || vec[i] <= 0 || vec[i] == CHAR_MAX)\), the size of the digit group is unlimited.

The required specializations return the empty string, indicating no grouping.

string_type do_truename() const;
string_type do_falsename() const;

5 Returns: A string representing the name of the boolean value true or false, respectively.

6 In the base class implementation these names are "true" and "false", or L"true" and L"false".

28.4.3.2 Class template numpunct_byname

namespace std {
    template<class charT>
    class numpunct_byname : public numpunct<charT> {
        // this class is specialized for char and wchar_t.
        public:
            using char_type = charT;
            using string_type = basic_string<charT>;

            explicit numpunct_byname(const char*, size_t refs = 0);
            explicit numpunct_byname(const string&, size_t refs = 0);

            protected:
            ~numpunct_byname();
        }
    }

28.4.4 The collate category

28.4.4.1 Class template collate

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;
        }
    }

\(^{268}\) 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".

\[\text{§ 28.4.4.1}\]
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;
};

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 (28.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.8.10).

Each function compares a string of characters `*p` in the range `[low, high)`.  

28.4.4.1 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)`.  

28.4.4.1.2 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 (28.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (25.8.10).

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.  

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. [Note: The probability that the result equals that for another string which does not compare equal should be very small, approaching `(1.0/numeric_limits<unsigned long>::max())`. —end note]

28.4.4.2 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);

269) This function is useful when one string is being compared to many other strings.

§ 28.4.4.2
protected:
  ~collate_byname();
};

28.4.5 The time category

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 28.4.5 only apply to the specializations required in Tables 102 and 103 (28.3.1.1.1). Their
members use their ios_base&, ios_base::iostate&, and fill arguments as described in 28.4, and the
cctype<> facet, to determine formatting details.

28.4.5.1 Class template time_get

namespace std {
  class time_base {
  public:
    enum dateorder { no_order, dmy, mdy, ymd, ydm };
  }

  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(); } 

    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,
                  const char_type* format, const char_type* modifier = 0) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                  const char_type* fmt, const char_type* fmtend) const;
  
    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;
  };

§ 28.4.5.1 1342
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{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.}

If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_base::eofbit in err.

### 28.4.5.1.1 Members

[locale.time.get.members]

```cpp
dateorder date_order() const;
Returns: do_date_order().
```

```cpp
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).
```

```cpp
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).
```

```cpp
iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
   ios_base::iostate& err, tm* t) const;
iter_type get_monthname(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).
```

```cpp
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).
```

```cpp
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).
```

```cpp
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.

Effects: The function starts by evaluating err = 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 fmt == fmtend evaluates to true.
2. The expression err == ios_base::goodbit evaluates to false.
3. The expression s == end evaluates to true, in which case the function evaluates err = ios_base::eofbit | ios_base::failbit.
4. The next element of fmt is equal to '%', optionally followed by a modifier character, followed by a conversion specifier character, format, together forming a conversion specification valid for the ISO/IEC 9945 function strptime. If the number of elements in the range [fmt, fmtend) is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates err = ios_base::failbit. Otherwise, the function evaluates s = do_get(s, end, f, err, t, format, modifier), where the value of modifier is '\0' when the optional modifier is absent from the conversion specification. If err == ios_base::goodbit holds after the evaluation of the expression, the function increments fmt to point just past the end of the conversion specification and continues looping.
The expression `isspace(*fmt, f.getloc())` evaluates to `true`, in which case the function first increments `fmt` until `fmt == fmtend || !isspace(*fmt, f.getloc())` evaluates to `true`, then advances `s` until `s == end || !isspace(*s, f.getloc())` is `true`, and finally resumes looping.

The next character read from `s` matches the element pointed to by `fmt` in a case-insensitive comparison, in which case the function evaluates `++fmt, ++s` and continues looping. Otherwise, the function evaluates `err = ios_base::failbit`.

[Note: The function uses the `ctype<charT>` facet installed in `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: `s`.

### 28.4.5.1.2 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.\(^{271}\) Returns `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 `struct tm` members, and remaining format characters, used by `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 `struct tm` members and remaining format characters used by `time_put<>::put` to produce one of the following formats, or until it encounters an error. The format depends on the value returned by `date_order()` as shown in Table 113.

<table>
<thead>
<tr>
<th><code>date_order()</code></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>

Table 113: `do_get_date` effects [tab:locale.time.get.dogetdate]

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.

#### `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.

\(^{271}\) This function is intended as a convenience only, for common formats, and may return `no_order` in valid locales.
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 strftime, 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.

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.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given format and modifier.

28.4.5.2 Class template time_get_byname

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;

      explicit time_get_byname(const char*, size_t refs = 0);
      explicit time_get_byname(const string&, size_t refs = 0);

    protected:
      ~time_get_byname();
    }
  }

28.4.5.3 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, 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;
  }
}
static locale::id id;

protected:
  ~time_put();
  virtual iter_type do_put(iter_type s, ios_base& str, char_type fill, const tm* t, char format, char modifier) const;
};

28.4.5.3.1 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: The fill argument may 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.

28.4.5.3.2 Virtual functions

iter_type do_put(iter_type s, ios_base& str, 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 specifiers in the string argument to the standard library function strftime(), except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.

Returns: An iterator pointing immediately after the last character produced. [Note: The fill argument may be used in the implementation-defined formats or by derivations. A space character is a reasonable default for this argument. — end note]

28.4.5.4 Class template 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();
  };
}

272) Although the C programming language defines no modifiers, most vendors do.
273) Interpretation of the modifier argument is implementation-defined, but should follow POSIX conventions.
274) Implementations should refer to other standards such as POSIX for these definitions.
28.4.6 The monetary category

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 money_put and money_get in the subclauses of 28.4.6 only apply to the specializations required in Tables 102 and 103 (28.3.1.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in 28.4, and the moneypunct<> and ctype<> facets, to determine formatting details.

28.4.6.1 Class template 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;
    };
}

28.4.6.1.1 Members

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;

Returns: do_get(s, end, intl, f, err, quant).

28.4.6.1.2 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: 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.

2 Where `money_base::space` or `money_base::none` appears as the last element in the format pattern, no white space is consumed. Otherwise, where `money_base::space` appears in any of the initial elements of the format pattern, at least one white space 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.

3 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: 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.

4 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

```c
for (int i = 0; i < n; ++i)
    buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);
```

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

```c
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.

28.4.6.2 Class template `money_put` [locale.money.put]

```c
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;
};

28.4.6.2.1 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).

28.4.6.2.2 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

c.t.widen(buf1, buf1 + sprintf(buf1, "%.0Lf", units), buf2)
for character buffers buf1 and buf2. If the first character in digits or buf2 is equal to c.t.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).

2 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: 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]

Returns: An iterator pointing immediately after the last character produced.

28.4.6.3 Class template moneypunct

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_symbol() const;
}
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`th component of the format. In the `field` member of 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, white space is permitted in the format, except where `none` appears at the end, in which case no white space 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:

```
value:
  units fractional_opt
  decimal-point digits
fractional:
  decimal-point digits_opt
```

if `frac_digits()` returns a positive value, or

```
value:
  units
```

otherwise. The symbol `decimal-point` indicates the character returned by `decimal_point()`. The other symbols are defined as follows:

```
units:
  digits
  digits thousands-sep units
digits:
  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()`.

---

276) An array of `char`, rather than an array of `part`, is specified for `pattern::field` purely for efficiency.
The placement of thousands-separator characters (if any) is determined by the value returned by `grouping()`, defined identically as the member `numpunct<>::do_grouping()`.

### 28.4.6.3.1 Members

```cpp
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;
```

Each of these functions `F` returns the result of calling the corresponding virtual member function `do_F()`.

### 28.4.6.3.2 Virtual functions

```cpp
charT do_decimal_point() const;

Returns: The radix separator to use in case `do_frac_digits()` is greater than zero.\(^{277}\)
```

```cpp
charT do_thousands_sep() const;

Returns: The digit group separator to use in case `do_grouping()` specifies a digit grouping pattern.\(^{278}\)
```

```cpp
string do_grouping() const;

Returns: A pattern defined identically as, but not necessarily equal to, the result of `numpunct<charT>::do_grouping()`.\(^{279}\)
```

```cpp
string_type do_curr_symbol() const;

Returns: A string to use as the currency identifier symbol. [Note: 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]
```

```cpp
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;\(^{280}\) `do_negative_sign()` returns the string to use to indicate a negative value.
```

```cpp
int do_frac_digits() const;

Returns: The number of digits after the decimal radix separator, if any.\(^{281}\)
```

```cpp
pattern do_pos_format() const;
pattern do_neg_format() const;

Returns: The specializations required in Table 103 (28.3.1.1.1), namely
```

\(^{277}\) In common U.S. locales this is ".".
\(^{278}\) In common U.S. locales this is ",".
\(^{279}\) To specify grouping by 3s, the value is "\003" not "3".
\(^{280}\) This is usually the empty string.
\(^{281}\) In common U.S. locales, this is 2.
\(^{282}\) Note that the international symbol returned by `do_curr_symbol()` usually contains a space, itself, for example, "USD ".

\[\text{§ 28.4.6.3.2}\]
28.4.6.4 Class template 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();
    };
}

28.4.7 The message retrieval category

Class messages<charT> implements retrieval of strings from message catalogs.

28.4.7.1 Class template messages

namespace std {
    class messages_base {
        public:
            using catalog = unspecified signed integer type;
    };

    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 cat, int set, intmsgid,
                            const string_type& dfault) const;
            void close(catalog cat) const;

            static locale::id id;

        protected:
            ~messages();
            virtual catalog do_open(const string&, const locale&) const;
            virtual string_type do_get(catalog, int set, intmsgid,
                                         const string_type& dfault) const;
            virtual void do_close(catalog) const;
    };
}

1 Values of type messages_base::catalog usable as arguments to members get and close can be obtained only by calling member open.

28.4.7.1.1 Members

catalog open(const string& name, const locale& loc) const;

1 Returns: do_open(name, loc).

string_type get(catalog cat, int set, intmsgid, const string_type& dfault) const;

2 Returns: do_get(cat, set,msgid, dfault).

void close(catalog cat) const;

3 Effects: Calls do_close(cat).
### 28.4.7.1.2 Virtual functions

```cpp
 catalog do_open(const string& name, const locale& loc) const;
```

1. **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()`.
2. Returns a value less than 0 if no such catalog can be opened.
3. **Remarks:** The locale argument `loc` is used for character set code conversion when retrieving messages, if needed.

```cpp
 string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const;
```

4. **Preconditions:** `cat` is a catalog obtained from `open()` and not yet closed.
5. **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`.

```cpp
 void do_close(catalog cat) const;
```

6. **Preconditions:** `cat` is a catalog obtained from `open()` and not yet closed.
7. **Effects:** Releases unspecified resources associated with `cat`.
8. **Remarks:** The limit on such resources, if any, is implementation-defined.

### 28.4.7.2 Class template messagesbyname

```cpp
 namespace std {
  template<class charT>
  class messagesbyname : public messages<charT> {
   public:
    using catalog = messagesbase::catalog;
    using string_type = basic_string<charT>;

    explicit messagesbyname(const char*, size_t refs = 0);
    explicit messagesbyname(const string&, size_t refs = 0);

    protected:
     ~messagesbyname();
  };
}
```

### 28.5 C library locales

#### 28.5.1 Header `<clocale>` synopsis

```cpp
 namespace std {
  struct lconv;

t char* setlocale(int category, const char* locale);
lconv* localeconv();
}
```

1. The contents and meaning of the header `<clocale>` are the same as the C standard library header `<locale.h>`.
2. Calls to the function `setlocale` may introduce a data race (16.5.5.10) 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 [tab: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>iswpunct</td>
<td>setlocale</td>
<td>wcscoll</td>
</tr>
<tr>
<td>iscntrl</td>
<td>iswalnum</td>
<td>iswpunct</td>
<td>strcoll</td>
<td>wcstod</td>
</tr>
<tr>
<td>isdigit</td>
<td>iswblank</td>
<td>iswupper</td>
<td>strerror</td>
<td>wcstombs</td>
</tr>
<tr>
<td>isgraph</td>
<td>iswcntrl</td>
<td>isxdigit</td>
<td>strtod</td>
<td>wcxfrm</td>
</tr>
<tr>
<td>islower</td>
<td>iswctype</td>
<td>isxdigit</td>
<td>strxfrm</td>
<td>wctomb</td>
</tr>
</tbody>
</table>
29  Input/output library [input.output]

29.1  General [input.output.general]

This Clause describes components that C++ programs may use to perform input/output operations.

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</td>
<td>Requirements</td>
</tr>
<tr>
<td>29.3</td>
<td>Forward declarations &lt;iosfwd&gt;</td>
</tr>
<tr>
<td>29.4</td>
<td>Standard iostream objects &lt;iostream&gt;</td>
</tr>
<tr>
<td>29.5</td>
<td>Iostreams base classes &lt;ios&gt;</td>
</tr>
<tr>
<td>29.6</td>
<td>Stream buffers &lt;streambuf&gt;</td>
</tr>
<tr>
<td>29.7</td>
<td>Formatting and manipulators &lt;iomanip&gt;, &lt;istream&gt;, &lt;ostream&gt;</td>
</tr>
<tr>
<td>29.8</td>
<td>String streams &lt;sstream&gt;</td>
</tr>
<tr>
<td>29.9</td>
<td>File streams &lt;fstream&gt;</td>
</tr>
<tr>
<td>29.10</td>
<td>Synchronized output streams &lt;synostream&gt;</td>
</tr>
<tr>
<td>29.11</td>
<td>File systems &lt;filesystem&gt;</td>
</tr>
<tr>
<td>29.12</td>
<td>C library files &lt;cstdio&gt;, &lt;cinttypes&gt;</td>
</tr>
</tbody>
</table>

[Note: 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.]

29.2  Iostreams requirements [iostreams.requirements]

29.2.1  Imbue limitations [iostream.limits.imbue]

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  

1 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.

2 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

1 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: Data races result in undefined behavior (6.9.2). — end note]

2 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_istringstream;
    template<class charT, class traits = char_traits<charT>,
            class Allocator = allocator<charT>>
        class basic_ostringstream;
    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;
}
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 Allocator = allocator<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_ostringstream<char>;
using iostream = basic_iostream<char>;

using stringbuf = basic_stringbuf<char>;
using istream = basic_istream<wchar_t>;
using ostream = basic_ostringstream<wchar_t>;
using wiostream = basic_iostream<wchar_t>;

using filebuf = basic_filebuf<char>;
using ifstream = basic_ifstream<char>;
using ofstream = basic_ofstream<char>;
using wfstream = basic_fstream<wchar_t>;

using synbuf = basic_syncbuf<char>;
using osyncstream = basic_osyncstream<char>;

using vstreambuf = basic_streambuf<wchar_t>;
using vistream = basic_istream<wchar_t>;
using vostream = basic_ostringstream<wchar_t>;

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.\footnote{283}
### 29.3.2 Overview

The class template specialization `basic_istream<charT, traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_istream` 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: 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: The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively. —end example]

[Note: 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 <istream> // 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 `<cstdio>` (29.12.1).

The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>`, 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. 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.

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.

---

284) If it is possible for them to do so, implementations should initialize the objects earlier than required.
285) 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`. 
Concurrent access to a synchronized (29.5.3.4) standard iostream object’s formatted and unformatted input (29.7.4.1) and output (29.7.5.1) functions or a standard C stream by multiple threads does not result in a data race (6.9.2). [Note: Users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid 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
```

```cpp
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 & noshownpoint (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.  

286) Typically long long.
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.  

29.5.3 Class ios_base

namespace std {

class ios_base {

class failure; // see below

// 29.5.3.1.2, 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;

// 29.5.3.1.3, 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;

// 29.5.3.1.4, 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;

// 29.5.3.1.5, seekdir
using seekdir = T4;
static constexpr seekdir beg = unspecified;
static constexpr seekdir cur = unspecified;
static constexpr seekdir end = unspecified;

class Init;

// 29.5.3.2, fmtflags state
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl);
fmtflags setf(fmtflags fmtfl, fmtflags mask);
void unsetf(fmtflags mask);

287) streamsize is used in most places where ISO C would use size_t.
streamsize precision() const;
streamsize precision(streamsize prec);
streamsize width() const;
streamsize width(streamsize wide);

// 29.5.3.3, locales
locale imbue(const locale& loc);
locale getloc() const;

// 29.5.3.5, storage
static int xalloc();
long& iword(int idx);
void*& pword(int idx);

// destructor
virtual ~ios_base();

// 29.5.3.6, callbacks
enum event { erase_event, imbue_event, copyfmt_event };
using event_callback = void (*)(event, ios_base&, int idx);
void register_callback(event_callback fn, int idx);

ios_base(const ios_base&) = delete;
ios_base& operator=(const ios_base&) = delete;
static bool sync_with_stdio(bool sync = true);

protected:
ios_base();

private:
static int index; // exposition only
long* iarray; // exposition only
void** parray; // exposition only
};

1 ios_base defines several member types:
   (1.1) — a type failure, defined as either a class derived from system_error or a synonym for a class derived from system_error;
   (1.2) — a class Init;
   (1.3) — three bitmask types, fmtflags, iostate, and openmode;
   (1.4) — an enumerated type, seekdir.

2 It maintains several kinds of data:
   (2.1) — state information that reflects the integrity of the stream buffer;
   (2.2) — control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;
   (2.3) — additional information that is stored by the program for its private use.

3 [Note: For the sake of exposition, the maintained data is presented here as:
   (3.1) — 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;
   (3.2) — long* iarray, points to the first element of an arbitrary-length long array maintained for the private use of the program;
   (3.3) — void** parray, points to the first element of an arbitrary-length pointer array maintained for the private use of the program.
   — end note]
29.5.3.1 Types

29.5.3.1.1 Class `ios_base::failure`

```c++
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: 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: 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]

```c++
explicit failure(const string& msg, const error_code& ec = io_errc::stream);
```

3. Effects: Constructs the base class with `msg` and `ec`.

```c++
explicit failure(const char* msg, const error_code& ec = io_errc::stream);
```

4. Effects: Constructs the base class with `msg` and `ec`.

29.5.3.1.2 Type `ios_base::fmtflags`

```c++
using fmtflags = T1;
```

1. The type `fmtflags` is a bitmask type (16.4.2.2.4). Setting its elements has the effects indicated in Table 116.

<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>showpos</td>
<td>generates a + sign in non-negative generated numeric 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>

2. Type `fmtflags` also defines the constants indicated in Table 117.
Table 117: fmtflags constants

<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>

29.5.3.1.3 Type ios_base::iostate

using iostate = T2;

1 The type `iostate` is a bitmask type (16.4.2.2.4) that contains the elements indicated in Table 118.

Table 118: iostate effects

<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>

2 Type `iostate` also defines the constant:

(2.1) — *goodbit*, the value zero.

29.5.3.1.4 Type ios_base::openmode

using openmode = T3;

1 The type `openmode` is a bitmask type (16.4.2.2.4). It contains the elements indicated in Table 119.

Table 119: openmode effects

<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>ato</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>

29.5.3.1.5 Type ios_base::seekdir

using seekdir = T4;

1 The type `seekdir` is an enumerated type (16.4.2.2.3) that contains the elements indicated in Table 120.

Table 120: seekdir effects

<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>
29.5.3.1.6 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:

```cpp
— static int init_cnt, counts the number of constructor and destructor calls for class Init, initialized to zero.
```

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.

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(), wclog.flush().

29.5.3.2 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().

29.5.3.3 Functions

locale imbue(const locale& loc);
1   Effects: Calls each registered callback pair (fn, idx) (29.5.3.6) 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: The previous value of getloc().

29.5.3.4 Static members

static bool sync_with_stdio(bool sync = true);
1   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.
2   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.
3   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.288

29.5.3.5 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

---

288) 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[ 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 `copyfmt`, calling `iword` with the same index yields another reference to the same value. If the function fails and `*this` is a base class subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

`Returns:` On success `iarray[ idx ]`. On failure, a valid `long&` initialized to 0.

```cpp
void*& pword(int idx);
```

`Preconditions:` `idx` is a value obtained by a call to `xalloc`.

`Effects:` If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `parray[ 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 `copyfmt`, calling `pword` with the same index yields another reference to the same value. If the function fails and `*this` is a base class subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

`Returns:` On success `parray[ idx ]`. On failure a valid `void*&` initialized to 0.

`Remarks:` After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

### Callbacks

A callback function is typically associated with an event that occurs when a member function of a derived class is called. The callback function is registered with the class and is called before or after the derivation occurs.

```cpp
void register_callback(event_callback fn, int idx);
```

`Preconditions:` The function `fn` does not throw exceptions.

`Effects:` Registers the pair `(fn, idx)` such that during calls to `imbue()` (29.5.3.3), `copyfmt()`, or `~ios_base()` (29.5.3.7), the function `fn` is called with argument `idx`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

`Remarks:` Identical pairs are not merged. A function registered twice will be called twice.

### Constructors and Destructor

The constructor initializes the members of the class, and the destructor calls the registered callback pairs.

```cpp
ios_base();
```

`Effects:` Each `ios_base` member has an indeterminate value after construction. The object’s members shall be initialized by calling `basic_ios::init` before the object’s first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```cpp
~ios_base();
```

`Effects:` Calls each registered callback pair `(fn, idx)` (29.5.3.6) as `(fn)(erase_event, *this, idx)` at such time that any `ios_base` member function called from within `fn` has well-defined results.

### Class Template `fpos`

The `fpos` class template is used to store offsets within basic streams.

```cpp
namespace std {
    template<class stateT>
    class fpos {
    public:
        // 29.5.4.1, members
        stateT state() const;
        void state(stateT);
    private:
        stateT st; // exposition only
    };
}
```

289) An implementation is free to implement both the integer array pointed at by `iarray` and the pointer array pointed at by `parray` as sparse data structures, possibly with a one-element cache for each.

290) For example, because it cannot allocate space.

291) 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

1. 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_copy_assignable_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,

(1.1) — P refers to an instance of fpos,
(1.2) — p and q refer to values of type P or const P,
(1.3) — pl and q1 refer to modifiable lvalues of type P,
(1.4) — 0 refers to type streamoff, and
(1.5) — o refers to a value of type streamoff or const streamoff.

<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></td>
<td>Effects: Value-initializes the state object.</td>
</tr>
<tr>
<td>P p = o;</td>
<td>P</td>
<td></td>
<td>Postconditions: p == P(o)</td>
</tr>
<tr>
<td>P()</td>
<td>P</td>
<td>P(0)</td>
<td></td>
</tr>
<tr>
<td>P p;</td>
<td>P</td>
<td>P(p0);</td>
<td></td>
</tr>
<tr>
<td>0(p)</td>
<td>streamoff</td>
<td>converts to offset</td>
<td>P(0(p)) == p</td>
</tr>
<tr>
<td>p != q</td>
<td>convertible to bool</td>
<td>!(p == q)</td>
<td></td>
</tr>
<tr>
<td>p + o</td>
<td>P</td>
<td>+ offset</td>
<td>Remarks: With q1 = p + o; then: q1 - o == p</td>
</tr>
<tr>
<td>pl += o</td>
<td>P&amp;</td>
<td>+= offset</td>
<td>Remarks: With q1 = pl; before the +=, then: pl - o == q1</td>
</tr>
<tr>
<td>p - o</td>
<td>P</td>
<td>- offset</td>
<td>Remarks: With q1 = p - o; then: q1 + o == p</td>
</tr>
<tr>
<td>pl -= o</td>
<td>P&amp;</td>
<td>-= offset</td>
<td>Remarks: With q1 = pl; before the -=, then: pl + o == q1</td>
</tr>
<tr>
<td>o + p</td>
<td>convertible to P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p - q</td>
<td>streamoff</td>
<td>distance</td>
<td>P(o + p) == p + o</td>
</tr>
</tbody>
</table>

2. Stream operations that return a value of type traits::pos_type return P(0(-1)) 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>>
```
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_ios();

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_ios();

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_ios::init()` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rdbuf()</code></td>
<td><code>sb</code></td>
</tr>
<tr>
<td><code>tie()</code></td>
<td>0</td>
</tr>
<tr>
<td><code>rdstate()</code></td>
<td>goodbit if <code>sb</code> is not a null pointer, otherwise badbit.</td>
</tr>
<tr>
<td><code>exceptions()</code></td>
<td>goodbit</td>
</tr>
<tr>
<td><code>flags()</code></td>
<td>`skipws</td>
</tr>
<tr>
<td><code>width()</code></td>
<td>0</td>
</tr>
<tr>
<td><code>precision()</code></td>
<td>6</td>
</tr>
<tr>
<td><code>fill()</code></td>
<td><code>widen(' ')</code></td>
</tr>
<tr>
<td><code>getloc()</code></td>
<td>a copy of the value returned by <code>locale()</code></td>
</tr>
<tr>
<td><code>iarray</code></td>
<td>a null pointer</td>
</tr>
<tr>
<td><code>parray</code></td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

29.5.5.3 Member functions

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)` (29.5.3.3) and if `rdbuf()` != 0 then `rdbuf()->pubimbue(loc)`
(29.6.3.2.1).

Returns: The prior value of `ios_base::imbue()`.

char narrow(char_type c, char dfault) const;

Returns: `use_facet<cwidth<char_type>>(getloc()).narrow(c, dfault)`
**char_type** widen(char c) const;

*Returns:* use_facet<ctype<char_type>>(getloc()).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:

1. calls each registered callback pair (fn, idx) as (*fn)(erase_event, *this, idx);
2. then, assigns to the member objects of *this the corresponding member objects of rhs, except that
   - rdstate(), rdbuf(), and exceptions() are left unchanged;
3. then, assigns to the member objects of *this the corresponding member objects of rhs, except that
   - the contents of arrays pointed at by pword and iword are copied, not the pointers themselves;
   - 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;
4. then, calls each callback pair that was copied from rhs as (*fn)(copyfmt_event, *this, idx);
5. then, calls exceptions(rhs.exceptions()).

*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.

---

292 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

[iostate.flags]

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);

Postconditions: If rdbuf() != 0 then state == rdstate(); otherwise rdstate() == (state | ios_base::badbit).

Effects: If ((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0, returns. Otherwise, the function throws an object of class ios_base::failure (29.5.3.1.1), constructed with implementation-defined argument values.

void setstate(iostate state);

Effects: Calls clear(rdstate() | state) (which may throw ios_base::failure (29.5.3.1.1)).

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);

Postconditions: except == exceptions().

Effects: Calls clear(rdstate()).

29.5.6 ios_base manipulators

[std.ios.manip]

29.5.6.1 fmtflags manipulators

[fmtflags.manip]

Each function specified in this subclause is a designated addressable function (16.5.4.2.1).

293) 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& no_showbase(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:** str.
Returns: str.

29.5.6.2 adjustfield manipulators [adjustfield.manip]
Each function specified in this subclause is a designated addressable function (16.5.4.2.1).

ios_base& internal(ios_base& str);
2 Effects: Calls str.setf(ios_base::internal, ios_base::adjustfield).
3 Returns: str.

ios_base& left(ios_base& str);
4 Effects: Calls str.setf(ios_base::left, ios_base::adjustfield).
5 Returns: str.

ios_base& right(ios_base& str);
6 Effects: Calls str.setf(ios_base::right, ios_base::adjustfield).
7 Returns: str.

29.5.6.3 basefield manipulators [basefield.manip]
Each function specified in this subclause is a designated addressable function (16.5.4.2.1).

ios_base& dec(ios_base& str);
2 Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
3 Returns: str.

ios_base& hex(ios_base& str);
4 Effects: Calls str.setf(ios_base::hex, ios_base::basefield).
5 Returns: str.

ios_base& oct(ios_base& str);
6 Effects: Calls str.setf(ios_base::oct, ios_base::basefield).
7 Returns: str.

29.5.6.4 floatfield manipulators [floatfield.manip]
Each function specified in this subclause is a designated addressable function (16.5.4.2.1).

ios_base& fixed(ios_base& str);
2 Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).
3 Returns: str.

ios_base& scientific(ios_base& str);
4 Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).
5 Returns: str.

ios_base& hexfloat(ios_base& str);
6 Effects: Calls str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield).
7 Returns: str.

8 [Note: The more obvious use of ios_base::hex to specify hexadecimal floating-point format would change the meaning of existing well-defined programs. C++ 2003 gives no meaning to the combination of fixed and scientific. — end note]

ios_base& defaultfloat(ios_base& str);
9 Effects: Calls str.unsetf(ios_base::floatfield).
10 Returns: str.

1374

§ 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()).

error_condition make_error_condition(io_errc e) noexcept;
2
   Returns: error_condition(static_cast<int>(e), iostream_category()).

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 \texttt{xnext} is not a null pointer and \texttt{xnext < xend} for an input sequence, then a \textit{read position} is available. In this case, \texttt{*xnext} shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

\textbf{29.6.3 Class template basic\_streambuf}

\begin{verbatim}
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_streambuf {
        public:
            template<class charT, class traits = char_traits<charT>>
            class basic_streambuf {
                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;

                public:
                    virtual ~basic_streambuf();

                    // 29.6.3.2.1, locales
                    locale pubimbue(const locale& loc);
                    locale getloc() const;

                    // 29.6.3.2.2, 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
                    // 29.6.3.2.3, get area
                    streamsize in_avail();
                    int_type snextc();
                    int_type sbumpc();
                    int_type sgetc();
                    streamsize sgetn(char_type* s, streamsize n);

                    // 29.6.3.2.4, putback
                    int_type sputbackc(char_type c);
                    int_type sungetc();

                    // 29.6.3.2.5, 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);

                        // 29.6.3.3.2, 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);

                        // 29.6.3.3.3, put area access
                        char_type* pbase() const;
        }
    }
}
\end{verbatim}
The class template `basic_streambuf` serves as an abstract base class for deriving various stream buffers whose objects each control two character sequences:

1. (1.1) — a character input sequence;
2. (1.2) — a character output sequence.

### 29.6.3.1 Constructors [streambuf.cons]

```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()`
5. (3.5) — `pptr()` == rhs.`pptr()`
6. (3.6) — `epptr()` == rhs.`epptr()`

295) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.
© ISO/IEC N4861

— getloc() == rhs.getloc()

  ~basic_streambuf();

  Effects: None.

29.6.3.2 Public member functions

29.6.3.2.1 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.

29.6.3.2.2 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().

29.6.3.2.3 Get area

streamsize in_avail();

  Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc() (29.6.3.4.3).

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).
29.6.3.2.4 Putback

\[\text{streambuf.pub.pback}\]

\begin{verbatim}
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()).
\end{verbatim}

\begin{verbatim}
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()).
\end{verbatim}

29.6.3.2.5 Put area

\[\text{streambuf.pub.put}\]

\begin{verbatim}
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).
\end{verbatim}

\begin{verbatim}
streamsize sputn(const char_type* s, streamsize n);

Returns: xsputn(s, n).
\end{verbatim}

29.6.3.3 Protected member functions

\[\text{streambuf.protected}\]

29.6.3.3.1 Assignment

\[\text{streambuf.assign}\]

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
void swap(basic_streambuf& rhs);

Effects: Swaps the data members of rhs and *this.
\end{verbatim}

29.6.3.3.2 Get area access

\[\text{streambuf.get.area}\]

\begin{verbatim}
char_type* eback() const;

Returns: The beginning pointer for the input sequence.
\end{verbatim}

\begin{verbatim}
char_type* gptr() const;

Returns: The next pointer for the input sequence.
\end{verbatim}

\begin{verbatim}
char_type* egptr() const;

Returns: The end pointer for the input sequence.
\end{verbatim}

\begin{verbatim}
void gbump(int n);

Effects: Adds n to the next pointer for the input sequence.
\end{verbatim}

\begin{verbatim}
void setg(char_type* gbeg, char_type* gnext, char_type* gend);

Postconditions: gbeg == eback(), gnext == gptr(), and gend == egptr() are all true.
\end{verbatim}
29.6.3.3 Put area access

char_type* pbase() const;

Returns: The beginning pointer for the output sequence.

char_type* pptr() const;

Returns: The next pointer for the output sequence.

char_type* epptr() const;

Returns: The end pointer for the output sequence.

void pbump(int n);

Effects: Adds n to the next pointer for the output sequence.

void setp(char_type* pbeg, char_type* pend);

Postconditions: pbeg == pbase(), pbeg == pptr(), and pend == epptr() are all true.

29.6.3.4 Virtual functions

29.6.3.4.1 Locales

void imbue(const locale&);

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.

29.6.3.4.2 Buffer management and positioning

basic_streambuf* setbuf(char_type* s, streamsize n);

Effects: Influences stream buffering in a way that is defined separately for each class derived from basic_streambuf in this Clause (29.8.2.4, 29.9.2.4).

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);

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 (29.8.2.4, 29.9.2.4).

Default behavior: Returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out);

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();

Effects: Synchronizes the controlled sequences with the arrays. That is, if pbase() is non-null the characters between pbase() and 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 (29.9.2.4).

Default behavior: Returns zero.
§ 29.6.3.4.3  Get area

streamsize showmanyc();

1  *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.

2  *Default behavior:* Returns zero.

3  *Remarks:* Uses *traits::eof()*.

streamsize xsgetn(char_type* s, streamsize n);

4  *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()*.

5  Assigning stops when either *n* characters have been assigned or a call to *sbumpc()* would return *traits::eof()*.

6  *Returns:* The number of characters assigned.

7  *Remarks:* Uses *traits::eof()*.

int_type underflow();

8  *Remarks:* The public members of *basic_streambuf* call this virtual function only if *gptr()* is null or *gptr() >= egptr()*.

9  *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.

10  The pending sequence is defined as the concatenation of

11  — the empty sequence if *gptr()* is null, otherwise the characters in [*gptr(), *egptr()*], followed by

12  — some (possibly empty) sequence of characters read from the input sequence.

13  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.

14  The backup sequence is the empty sequence if *eback()* is null, otherwise the characters in [*eback(), *gptr()*].

15  *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()*.

16  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

17  — 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

18  — the characters in [*gptr() - n, *gptr()*] agree with the backup sequence (where *n* is the length of the backup sequence).

19  *Default behavior:* Returns *traits::eof()*.

int_type uflow();

20  *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.

---

296) The morphemes of *showmanyc* are "es-how-many-see", not "show-manic".

297) *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".

298) 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.

### 29.6.3.4.4 Putback

```cpp
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

- 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.
- 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.

Postconditions: On return, the constraints of `gptr()`, `eback()`, and `pptr()` are the same as for `underflow()`.

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()`.

### 29.6.3.4.5 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

- the empty sequence if `pbase()` is null, otherwise the `pptr() - pbase()` characters beginning at `pbase()`, followed by
- 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:

- The effect of consuming a character on the associated output sequence is specified.\(^{299}\)
- 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.

---

\(^{299}\) 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>
    basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);
    template<class charT, class traits, class T>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& 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>
    basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);
    template<class charT, class traits>
    basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>& os);
    template<class charT, class traits>
    basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);
    template<class charT, class traits>
    basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);
    template<class charT, class traits>
    basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);
    template<class charT, class traits>
    basic_ostream<charT, traits>& flush_emit(basic_ostream<charT, traits>& os);
    template<class charT, class traits, class T>
    basic_ostream<charT, traits>& operator<<((basic_ostream<charT, traits>&& os, const T& x);
}
```

#### 29.7.3 Header `<iomanip>` synopsis

```cpp
namespace std {
    // types T1, T2, ... are unspecified implementation types
    T1 resetiosflags(ios_base::fmtflags mask);
}
```

---

300) 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)`.

---

§ 29.7.3 1383
29.7.4 Input streams [input.streams]

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.

### 29.7.4.1 Class template `basic_istream`

```cpp
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;

        // 29.7.4.1.1, constructor/destructor
        explicit basic_istream(basic_streambuf<charT, traits>* sb);
        virtual ~basic_istream();

        // 29.7.4.1.3, prefix/suffix
        class sentry;

        // 29.7.4.2, formatted input
        basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&&));
        basic_istream<charT, traits>& operator>>(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
        basic_istream<charT, traits>& operator>>(ios_base& (*pf)(ios_base&));

        basic_istream<charT, traits>& operator>>(bool& n);
        basic_istream<charT, traits>& operator>>(short& n);
        basic_istream<charT, traits>& operator>>(unsigned short& n);
        basic_istream<charT, traits>& operator>>(int& n);
        basic_istream<charT, traits>& operator>>(unsigned int& n);
        basic_istream<charT, traits>& operator>>(long& 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.

### 29.7.4.1.1 Constructors

```cpp
class basic_istream
{
public:
  explicit basic_istream(basic_streambuf<charT, traits>* sb);
  basic_istream(basic_istream&& rhs);
  virtual ~basic_istream();
};
```

1. **Effects:** Initializes the base class subobject with `basic_ios::init(sb)` (29.5.5.2).
2. **Postconditions:** `gcount()` == 0.
3. **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.
4. **Remarks:** Does not perform any operations of `rdbuf()`.

### 29.7.4.1.2 Assignment and swap

```cpp
basic_istream& operator=(basic_istream&& rhs);

virtual ~basic_istream();
```

1. **Effects:** Equivalent to: `swap(rhs)`.
2. **Returns:** *this.
3. **Effects:** Calls `basic_ios<charT, traits>::swap(rhs)`. Exchanges the values returned by `gcount()` and `rhs.gcount()`.

### 29.7.4.1.3 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;
  };
}
```

1. The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.
2. **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

---

© ISO/IEC N4861
the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.\(^{301}\) 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`).\(^{302}\)

\[\text{~sentry}();\]

Effects: None.

```cpp
explicit operator bool() const;
```

Returns: `ok_`.

29.7.4.2 Formatted input functions \([\text{istream.formatted}]\)

29.7.4.2.1 Common requirements \([\text{istream.formatted.reqmts}]\)

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\(^{303}\) 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`.

29.7.4.2.2 Arithmetic extractors \([\text{istream.formatted.arithmetic}]\)

```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<>` (28.4.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 29.7.4.2.1). 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);
```

\(^{301}\) 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.

\(^{302}\) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

\(^{303}\) 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: 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`.

```c++
operator>>(short& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```c++
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);
```

```c++
operator>>(int& val);
```

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```c++
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);
```

29.7.4.2.3 basic_istream::operator>> [istream.extractors]

```c++
basic_istream<charT, traits>&
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 29.7.4.2.1).

2 Returns: `pf(*this)`.304

```c++
basic_istream<charT, traits>&
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 29.7.4.2.1).

4 Returns: `*this`.

```c++
basic_istream<charT, traits>& operator>>(ios_base& (*pf)(ios_base&));
```

5 Effects: Calls `pf(*this)`.305 This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

6 Returns: `*this`.

304 See, for example, the function signature `ws(basic_istream&)` (29.7.4.4).
305 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 29.7.4.2.1) of in. After a sentry object is constructed, \texttt{operator\textgreater\textless} extracts characters and stores them into \texttt{s}. If \texttt{width()} is greater than zero, \text{n} is \text{min(}size\texttt{t}(\texttt{width}()), N). Otherwise \text{n} is N. \text{n} is the maximum number of characters stored.

Characters are extracted and stored until any of the following occurs:

1. \text{n}-1 characters are stored;
2. end of file occurs on the input sequence;
3. \texttt{operator\textgreater\textless} then stores a null byte (\texttt{charT()}) in the next position, which may be the first position if no characters were extracted. \texttt{operator\textgreater\textless} then calls \texttt{width(0)}.
4. If the function extracted no characters, it calls \texttt{setstate(failbit)}, which may throw \texttt{ios\_base::failure (29.5.5.4)}.

Returns: \texttt{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 29.7.4.2.1) of \texttt{in}. After a sentry object is constructed a character is extracted from \texttt{in}, if one is available, and stored in \texttt{c}. Otherwise, the function calls \texttt{in.setstate(failbit)}.

Returns: \texttt{in}.

basic_istream<charT, traits>& operator>>(basic_streambuf<charT, traits>* sb);

Effects: Behaves as an unformatted input function (29.7.4.3). If \texttt{sb} is null, calls \texttt{setstate(failbit)}, which may throw \texttt{ios\_base::failure (29.5.5.4)}. After a sentry object is constructed, extracts characters from \texttt{*this} and inserts them in the output sequence controlled by \texttt{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 \texttt{setstate(failbit)}, which may throw \texttt{ios\_base::failure (29.5.5.4)}. If it inserted no characters because it caught an exception thrown while extracting characters from \texttt{*this} and \texttt{failbit} is set in \texttt{exceptions()} (29.5.5.4), then the caught exception is rethrown.

Returns: \texttt{*this}.

29.7.4.3 Unformatted input functions

Each unformatted input function begins execution by constructing an object of class \texttt{sentry} with the default argument \texttt{noskipws} (second) argument \texttt{true}. If the \texttt{sentry} object returns \texttt{true}, when converted to a value of type \texttt{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 \texttt{false}, when converted to a value of type \texttt{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 \texttt{charT()}) in the first location of the array. If an exception is thrown during input then

\section*{§ 29.7.4.3 1389}
ios_base::badbit is turned on in *this’s error state. (Exceptions thrown from basic_ios<>::clear() 
are not caught or rethrown.) If (exceptions()&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.

```cpp
streamsize gcount() const;
2   Effects: None. This member function does not behave as an unformatted input function (as described 
3   above).
4   Returns: The number of characters extracted by the last unformatted input member function called for 
5   the object.
int_type get();
4   Effects: Behaves as an unformatted input function (as described above). After constructing a sentry 
5   object, extracts a character c, if one is available. Otherwise, the function calls setstate(failbit), 
6   which may throw ios_base::failure (29.5.5.4).
7   Returns: c if available, otherwise traits::eof().
```

```cpp
basic_istream<charT, traits>& get(char_type& c);
6   Effects: Behaves as an unformatted input function (as described above). After constructing a sentry 
7   object, extracts a character, if one is available, and assigns it to c. Note that this function is not 
8   overloaded on types signed char and unsigned char.
9   Returns: *this.
```

```cpp
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 
9   object, extracts characters and stores them into successive locations of an array whose first element is 
10  designated by s. Note that this function is not overloaded on types signed char and unsigned char.
11  Characters are extracted and stored until any of the following occurs:
12
13  — n is less than one or n - 1 characters are stored;
14  — end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit));
15  — traits::eq(c, delim) for the next available input character c (in which case c is not extracted).
16  If the function stores no characters, it calls setstate(failbit) (which may throw ios_base::failure 
17  (29.5.5.4)). In any case, if n is greater than zero it then stores a null character into the next successive 
18  location of the array.
19  Returns: *this.
```

```cpp
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.
```

```cpp
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 
14   object, extracts characters and inserts them in the output sequence controlled by sb. Characters are 
15   extracted and inserted until any of the following occurs:
16
17  — end-of-file occurs on the input sequence;
18  — inserting in the output sequence fails (in which case the character to be inserted is not extracted);
19  — traits::eq(c, delim) for the next available input character c (in which case c is not extracted);
20  — an exception occurs (in which case, the exception is caught but not rethrown).
21  If the function inserts no characters, it calls setstate(failbit), which may throw ios_base::failure 
22  (29.5.5.4).
```

---

306) This is done without causing an ios_base::failure to be thrown.
307) Note that this function is not overloaded on types signed char and unsigned char.
308) Note that this function is not overloaded on types signed char and unsigned char.
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.309 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 setstate(eofbit));
2. traits::eq(c, delim) for the next available input character c (in which case the input character is extracted but not stored);310
3. n is less than one or n - 1 characters are stored (in which case the function calls setstate(failbit)).

These conditions are tested in the order shown.311
If the function extracts no characters, it calls setstate(failbit) (which may throw ios_base::failure (29.5.5.4)).312
In any case, if n is greater than zero, it then stores a null character (using charT()) into the next successive location of the array.

Returns: *this.

[Example:
#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;
    }
}
—end example]

basic_istream<charT, traits>& getline(char_type* s, streamsize n);

Returns: getline(s, n, widen('\n'))

309) Note that this function is not overloaded on types signed char and unsigned char.
310) Since the final input character is "extracted", it is counted in the gcount(), even though it is not stored.
311) This allows an input line which exactly fills the buffer, without setting failbit. This is different behavior than the historical AT&T implementation.
312) This implies an empty input line will not cause 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:

(25.1) \( n \neq \text{numeric_limits}<\text{streamsize}>::\text{max()} \) (17.3.5) and \( n \) characters have been extracted so far

(25.2) end-of-file occurs on the input sequence (in which case the function calls \text{setstate(eofbit)}, which may throw \text{ios_base::failure} (29.5.5.4));

(25.3) \text{traits::eq_int_type(traitsto_int_type(c), delim)} for the next available input character \( c \) (in which case \( c \) is extracted).

Remarks: The last condition will never occur if \text{traits::eq_int_type(delim, traits::eof())}.

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: \text{traits::eof()} if \text{good()} is false. Otherwise, returns \text{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 \!\text{good()} calls \text{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:

(30.1) \( n \) characters are stored;

(30.2) end-of-file occurs on the input sequence (in which case the function calls \text{setstate(eofbit | eofbit)}, which may throw \text{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 \!\text{good()} calls \text{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 \text{rdbuf()}->\text{in_avail()} == -1, calls \text{setstate(eofbit)} (which may throw \text{ios_base::failure} (29.5.5.4)), and extracts no characters;

(32.1) If \text{rdbuf()}->\text{in_avail()} == 0, extracts no characters

(32.2) If \text{rdbuf()}->\text{in_avail()} > 0, extracts \text{min(rdbuf()}->\text{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 \!\text{good()} calls \text{setstate(failbit)} which may throw an exception, and return. If \text{rdbuf()} is not null, calls \text{rdbuf()}->\text{sputbackc}(c). If \text{rdbuf()} is null, or if \text{sputbackc} returns \text{traits::eof()}, calls \text{setstate(badbit)} (which may throw \text{ios_base::failure} (29.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to \text{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 \!\text{good()} calls \text{setstate(failbit)} which may throw an exception, and return. If \text{rdbuf()} is not null, calls \text{rdbuf()}->\text{sungetc()}. If \text{rdbuf()}

Note that this function is not overloaded on types \text{signed char} and \text{unsigned char}.
is null, or if `sungetc` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure (29.5.5.4)`). [Note: This function extracts no characters, so the value returned by the next call to `gcount()` is 0. — end note]

```
37
Returns: *this.
```

```plaintext
38
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 `ios_base::failure (29.5.5.4)`), and returns -1. Otherwise, returns zero.

```
39
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()`.

```
40
Returns: After constructing a sentry object, if `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, in)`. 
```

```
41
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()->pubseekpos(pos, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

```
42
Returns: *this.
```

```
43
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 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 `ios_base::failure`).

```
44
Returns: *this.
```

### 29.7.4.4 Standard basic_istream manipulators

Each instantiation of the function template specified in this subclause is a designated addressable function (16.5.4.2.1).

```
1
template<class charT, class traits>
basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);
```

**Effects:** Behaves as an unformatted input function (29.7.4.3), 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 `c` 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 (29.7.4.1.3)`. If `ws` stops extracting characters because there are no more available it sets `eofbit`, but not `failbit`.

```
2
Returns: is.
```

### 29.7.4.5 Rvalue stream extraction

```
1
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 `ios_base`.

**Effects:** Equivalent to:

```
1
is >> std::forward<T>(x);
```
return std::move(is);

29.7.4.6 Class template basic_iostream
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;

            // 29.7.4.6.1, constructor
            explicit basic_iostream(basic_streambuf<charT, traits>* sb);

            // 29.7.4.6.2, destructor
            virtual ~basic_iostream();

            protected:
                // 29.7.4.6.1, constructor
                basic_iostream(const basic_iostream&) = delete;
                basic_iostream(basic_iostream&& rhs);

                // 29.7.4.6.3, assign and swap
                basic_iostream& operator=(const basic_iostream&);
                basic_iostream& operator=(basic_iostream&& rhs);
                void swap(basic_iostream& rhs);
        } // basic_iostream
    }
}

The class template basic_iostream inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

29.7.4.6.1 Constructors
explicit basic_iostream(basic_streambuf<charT, traits>* sb);

Effects: Initializes the base class subobjects with basic_istream<charT, traits>(sb) (29.7.4.1) and basic_ostream<charT, traits>(sb) (29.7.5.1).

Postconditions: rdbuf() == sb and gcount() == 0.

29.7.4.6.2 Destructor
virtual ~basic_iostream();

Remarks: Does not perform any operations on rdbuf().

29.7.4.6.3 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
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.
29.7.5.1 Class template basic_ostream

```c++
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;

            // 29.7.5.1.1, constructor/destructor
            explicit basic_ostream(basic_streambuf<char_type, traits>* sb);
            virtual ~basic_ostream();

            // 29.7.5.1.3, prefix/suffix
            class sentry;

            // 29.7.5.2, 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 n);
            basic_ostream<charT, traits>& operator<<(unsigned long long n);
            basic_ostream<charT, traits>& operator<<(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);

            // 29.7.5.3, 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();

            // 29.7.5.1.4, seeks
            pos_type tellp();
            basic_ostream<charT, traits>& seekp(pos_type);
            basic_ostream<charT, traits>& seekp(off_type, ios_base::seekdir);

        protected:
            // 29.7.5.1.1, copy/move constructor
            basic_ostream(const basic_ostream&) = delete;
            basic_ostream(basic_ostream&& rhs);

            // 29.7.5.1.2, assign and swap
            basic_ostream& operator=(const basic_ostream&);
            basic_ostream& operator=(basic_ostream&& rhs);

    }
}
```
void swap(basic_ostream& rhs);
};

// 29.7.5.2.4, 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>&, signed char);

template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>&, unsigned char);

template<class traits>
basic_ostream<wchar_t, traits>& operator<<(basic_ostream<wchar_t, traits>&, wchar_t) = delete;

template<class traits>
basic_ostream<char8_t, traits>& operator<<(basic_ostream<char8_t, traits>&, char8_t) = delete;

template<class traits>
basic_ostream<char16_t, traits>& operator<<(basic_ostream<char16_t, traits>&, char16_t) = delete;

template<class traits>
basic_ostream<char32_t, traits>& operator<<(basic_ostream<char32_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 traits>
basic_ostream<wchar_t, traits>& operator<<(basic_ostream<wchar_t, traits>&, const wchar_t*) = delete;

template<class traits>
basic_ostream<char8_t, traits>& operator<<(basic_ostream<char8_t, traits>&, const char8_t*) = delete;

template<class traits>
basic_ostream<char16_t, traits>& operator<<(basic_ostream<char16_t, traits>&, const char16_t*) = delete;

template<class traits>
basic_ostream<char32_t, traits>& operator<<(basic_ostream<char32_t, traits>&, const char32_t*) = delete;
template<class traits>
    basic_ostream<wchar_t, traits>&
    operator<<(basic_ostream<wchar_t, traits>&, const char32_t*) = delete;
}

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.

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().

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.

[Note: The deleted overloads of operator<< prevent formatting characters as integers and strings as pointers. — end note]

29.7.5.1.1 Constructors [ostream.cons]

    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.

    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.

    virtual ~basic_ostream();
4 Remarks: Does not perform any operations on rdbuf().

29.7.5.1.2 Assignment and swap [ostream.assign]

    basic_ostream& operator=(basic_ostream&& rhs);
1 Effects: Equivalent to: swap(rhs).
2 Returns: *this.

    void swap(basic_ostream& rhs);
3 Effects: Calls basic_ios<charT, traits>::swap(rhs).

29.7.5.1.3 Class basic_ostream::sentry [ostream.sentry]

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_; }

        sentry(const sentry&)=delete;
        sentry& operator=(const sentry&)=delete;
    };
}

1 The class sentry defines a class that is responsible for doing exception safe prefix and suffix operations.
explicit sentry(basic_ostream<charT, traits>& os);
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 (29.5.5.4)).

~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.

explicit operator bool() const;
5
   Effects: Returns ok_.

29.7.5.1.4  Seek members
[ostream.seeks]
1
   Each seek member function begins execution by constructing an object of class sentry. It returns by
   destroying the sentry object.

   pos_type tellp();
2
      Returns: If fail() != false, returns pos_type(-1) to indicate failure. Otherwise, returns rdbuf()->
      pubseekoff(0, cur, out).

   basic_ostream<charT, traits>& seekp(pos_type pos);
3
      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).

      Returns: *this.

   basic_ostream<charT, traits>& seekp(off_type off, ios_base::seekdir dir);
4
      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).

      Returns: *this.

29.7.5.2  Formatted output functions
[ostream.formatted]
29.7.5.2.1  Common requirements
[ostream.formatted.reqmts]
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 might throw an exception. 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. 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.

   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 of
   os.width() characters. If (os.flags() & ios_base::adjustfield) == ios_base::left is true, the fill
   characters are placed after the character sequence; otherwise, they are placed before the character sequence.

29.7.5.2.2  Arithmetic inserters
[ostream.inserters.arithmetic]
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 ofstreams 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.

Returns: *this.
29.7.5.2.3 basic_ostream::operator<<

basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));

Effects: None. Does not behave as a formatted output function (as described in 29.7.5.2.1).

Returns: pf(*this).\[317\]

basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));

Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 29.7.5.2.1).

Returns: *this.\[318\]

basic_ostream<charT, traits>& operator<<(ios_base& (*pf)(ios_base&));

Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 29.7.5.2.1).

Returns: *this.

basic_ostream<charT, traits>& operator<<(basic_streambuf<charT, traits>* sb);

Effects: Behaves as an unformatted output function (29.7.5.3). After the sentry object is constructed, if sb is null calls setstate(badbit) (which may throw ios_base::failure).

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.

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.

Returns: *this.

basic_ostream<charT, traits>& operator<<(nullptr_t);

Effects: Equivalent to:

return *this << s;

where s is an implementation-defined NTCTS (16.3.18).

29.7.5.2.4 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);
template<class traits>
basic_ostream<char, traits>&
operator<<(basic_ostream<char, traits>& out, unsigned char c);

Effects: Behaves as a formatted output function (29.7.5.2.1) 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 29.7.5.2.1. Inserts seq into out. Calls os.width(0).

\[317\] See, for example, the function signature endl(basic_ostream&) (29.7.5.4).

\[318\] See, for example, the function signature dec(ios_base&) (29.5.6.3).
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 29.7.5.2.1) 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<char, 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 29.7.5.2.1. Inserts seq into out. Calls width(0).

Returns: out.

29.7.5.3 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;

(5.2) 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.

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

319) without causing an ios_base::failure to be thrown.
320) Note that this function is not overloaded on types signed char and unsigned char.
321) Note that this function is not overloaded on types signed char and unsigned char.
the function calls \texttt{rdbuf()->pubsync()}. If that function returns -1 calls \texttt{setstate(badbit)} (which may throw \texttt{ios\_base::failure (29.5.5.4))}. Otherwise, if the sentry object returns \texttt{false}, does nothing.

\textbf{Returns:} *this.

\subsection*{29.7.5.4 Standard manipulators \hfill [ostream.manip]}

Each instantiation of any of the function templates specified in this subclause is a designated addressable function (16.5.4.2.1).

\begin{verbatim}
template<class charT, class traits>
    basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);

Effects: Calls os.put(os.widen(\texttt{'\n'})), then os.flush().

Returns: os.

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.

template<class charT, class traits>
    basic_ostream<charT, traits>& emit_on_flush(basic_ostream<charT, traits>& os);

Effects: If os.rdbuf() is a \texttt{basic\_syncbuf<charT, traits, Allocator>*}, called buf for the purpose of exposition, calls buf->set_emit_on_sync(true). Otherwise this manipulator has no effect. \hspace{1em} [Note: To work around the issue that the Allocator template argument cannot be deduced, implementations can introduce an intermediate base class to \texttt{basic\_syncbuf} that manages its emit_on_sync flag. \hspace{1em} \textbf{-end note}]

Returns: os.

template<class charT, class traits>
    basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);

Effects: If os.rdbuf() is a \texttt{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.

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 \texttt{basic\_syncbuf<charT, traits, Allocator>*}, called buf for the purpose of exposition, calls buf->emit().

Returns: os.
\end{verbatim}

\subsection*{29.7.5.5 Rvalue stream insertion \hfill [ostream.rvalue]}

\begin{verbatim}
template<class Ostream, class T>
    Ostream&& operator<<(Ostream&& os, const T& x);

Constraints: The expression os \hspace{1em} <= x is well-formed when treated as an unevaluated operand and \texttt{Ostream} is publicly and unambiguously derived from \texttt{ios\_base}.

Effects: As if by: os << x;

Returns: std::move(os).
\end{verbatim}

\subsection*{29.7.6 Standard manipulators \hfill [std.manip]}

The header \texttt{iomanip} defines several functions that support extractors and inserters that alter information maintained by class \texttt{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:

```c++
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:

```c++
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:

```c++
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> then the expression out << setfill(c) behaves as if it called f(out, c), where the function f is defined as:

```c++
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.

322) 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 1403
unspecified 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:

```cpp
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.

unspecified setw(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 << 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:

```cpp
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

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.

```cpp
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 (29.7.4.2.1).

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:

```cpp
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 >> get_money(mon, intl) has type basic_istream<charT, traits>& and value in.

```cpp
template<class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);
```

Mandates: The type moneyT is either `long double` or a specialization of the basic_string template (Clause 21).

§ 29.7.7
Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << put_money(mon, intl)` behaves as a formatted output function (29.7.5.2.1) that calls `f(out, mon, intl)`, where the function `f` is defined as:

```cpp
template<class charT, class traits, class moneyT>
void f(basic_ostream<charT, traits>& str, const moneyT& mon, bool intl) {
    using Iter = ostreambuf_iterator<charT, traits>;
    using MoneyPut = money_put<charT, Iter>;
    const MoneyPut& mp = use_facet<MoneyPut>(str.getloc());
    const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);
    if (end.failed())
        str.setstate(ios_base::badbit);
}
```

The expression `out << put_money(mon, intl)` has type `basic_ostream<charT, traits>&` and value `out`.

```cpp
template<class charT> unspecified get_time(struct tm* tmb, const charT* fmt);
```

Preconditions: The argument `tmb` is a valid pointer to an object of type `struct tm`, and `[fmt, fmt + char_traits<charT>::length(fmt))` is a valid range.

Returns: An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >> get_time(tmb, fmt)` behaves as if it called `f(in, tmb, fmt)`, where the function `f` is defined as:

```cpp
template<class charT, class traits>
void f(basic_istream<charT, traits>& str, struct tm* tmb, const charT* fmt) {
    using Iter = istreambuf_iterator<charT, traits>;
    using TimeGet = time_get<charT, Iter>;
    ios_base::iostate err = ios_base::goodbit;
    const TimeGet& tg = use_facet<TimeGet>(str.getloc());
    tg.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
    fmt, fmt + traits::length(fmt));
    if (err != ios_base::goodbit)
        str.setstate(err);
}
```

The expression `in >> get_time(tmb, fmt)` has type `basic_istream<charT, traits>&` and value `in`.

```cpp
template<class charT> unspecified put_time(const struct tm* tmb, const charT* fmt);
```

Preconditions: The argument `tmb` is a valid pointer to an object of type `struct tm`, and `[fmt, fmt + char_traits<charT>::length(fmt))` is a valid range.

Returns: An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out << put_time(tmb, fmt)` behaves as if it called `f(out, tmb, fmt)`, where the function `f` is defined as:

```cpp
template<class charT, class traits>
void f(basic_ostream<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
    using Iter = ostreambuf_iterator<charT, traits>;
    using TimePut = time_put<charT, Iter>;
    const TimePut& tp = use_facet<TimePut>(str.getloc());
    const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
    fmt, fmt + traits::length(fmt));
    if (end.failed())
        str.setstate(ios_base::badbit);
}
```

§ 29.7.7
The expression `out << put_time(tmb, fmt)` has type `basicostream<charT, traits>&` and value `out`.

### 29.7.8 Quoted manipulators

1. 
   Note: 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

   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<charT, traits, Allocator>& s, charT delim = charT('"'), charT escape = charT('"'));

   template<class charT, class traits>
   unspecified quoted(basic_string_view<charT, traits> 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 (29.7.5.2.1) of `out`. This forms a character sequence `seq`, initially consisting of the following elements:

   - (2.1) `delim`.
   - (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`.
   - (2.3) `delim`.

   Let `x` be the number of elements initially in `seq`. Then padding is determined for `seq` as described in 29.7.5.2.1, `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`.

   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:

   - (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&)` (29.7.4.2.3) which may throw `ios_base::failure` (29.5.3.1.1):
     - (3.1.1) If the first character extracted is equal to `delim`, as determined by `traits_type::eq`, then:
       - (3.1.1.1) Turn off the `skipws` flag.
       - (3.1.1.2) `s.clear()`.
       - (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`.
       - (3.1.1.4) Discard the final `delim` character.
       - (3.1.1.5) Restore the `skipws` flag to its original value.
     - (3.1.2) Otherwise, `in >> s`.
   - (3.2) 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.
   - (3.3) The expression `in >> quoted(s, delim, escape)` has type `basic_istream<charT, traits>&` and value `in`. 

§ 29.7.8

© ISO/IEC N4861

1406
The expression `out << quoted(s, delim, escape)` has type `basic_ostream<charT, traits>&` and value `out`.

29.8 String-based streams

29.8.1 Header `<sstream>` synopsis

```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>;
}
```

1 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`

```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;

        // 29.8.2.1, 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);

template<class SAlloc>
    explicit basic_stringbuf(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which = ios_base::in | ios_base::out);

    basic_stringbuf(const basic_stringbuf&) = delete;
    basic_stringbuf(basic_stringbuf&& rhs);
    basic_stringbuf(basic_stringbuf&& rhs, const Allocator& a);

    // 29.8.2.2, assign and swap
    basic_stringbuf& operator=(const basic_stringbuf&) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf&& rhs) noexcept(see below);

    // 29.8.2.3, 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<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);

protected:
    // 29.8.2.4, 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
};

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)));

1 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.

2 For the sake of exposition, the maintained data and internal pointer initialization is presented here as:
(2.1) — `ios_base::openmode mode`, has in set if the input sequence can be read, and out set if the output sequence can be written.

(2.2) — `basic_string<charT, traits, Allocator> buf` contains the underlying character sequence.

(2.3) — `init_buf_ptrs()` sets the base class’ get area (29.6.3.3.2) and put area (29.6.3.3.3) pointers after initializing, moving from, or assigning to `buf` accordingly.

### 29.8.2.1 Constructors

**explicit basic_stringbuf(ios_base::openmode which);**

1. **Effects:** Initializes the base class with `basic_streambuf()` (29.6.3.1), 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);**

3. **Effects:** Initializes the base class with `basic_streambuf()` (29.6.3.1), `mode` with `which`, and `buf` with `s`, then calls `init_buf_ptrs()`.

**basic_stringbuf(ios_base::openmode which, const Allocator &a);**

4. **Effects:** Initializes the base class with `basic_streambuf()` (29.6.3.1), `mode` with `which`, and `buf` with `a`, then calls `init_buf_ptrs()`.

5. **Postconditions:** `str().empty()` is true.

**explicit basic_stringbuf(basic_string<charT, traits, Allocator>&& s, ios_base::openmode which = ios_base::in | ios_base::out);**

6. **Effects:** Initializes the base class with `basic_streambuf()` (29.6.3.1), `mode` with `which`, and `buf` with `{std::move(s)}, then calls `init_buf_ptrs()`.

**template<class SAlloc> basic_stringbuf(const basic_string<charT, traits, SAlloc>& s, ios_base::openmode which = ios_base::in | ios_base::out);**

7. **Effects:** Initializes the base class with `basic_streambuf()` (29.6.3.1), `mode` with `which`, and `buf` with `{s, a}`, then calls `init_buf_ptrs()`.

**template<class SAlloc> explicit basic_stringbuf(const basic_string<charT, traits, SAlloc>& s, ios_base::openmode which = ios_base::in | ios_base::out);**

8. **Constraints:** `is_same_v<SAlloc, Allocator>` is false.

9. **Effects:** Initializes the base class with `basic_streambuf()` (29.6.3.1), `mode` with `which`, and `buf` with `s`, then calls `init_buf_ptrs()`.

**basic_stringbuf(basic_stringbuf&& rhs); 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) \[ \text{pptr()} - \text{pbase()} == \text{rhs.p.pptr()} - \text{rhs.p.pbase()} \]
(11.5) \[ \text{eptr()} - \text{pbase()} == \text{rhs.p.eptr()} - \text{rhs.p.pbase()} \]
(11.6) \[ \text{if (eback()) eback()} != \text{rhs.a.eback()} \]
(11.7) \[ \text{if (gptr()) gptr()} != \text{rhs.a.gptr()} \]
(11.8) \[ \text{if (egptr()) egptr()} != \text{rhs.a.egptr()} \]
(11.9) \[ \text{if (pbase()) pbase()} != \text{rhs.a.pbase()} \]
(11.10) \[ \text{if (pptr()) pptr()} != \text{rhs.a.pptr()} \]
(11.11) \[ \text{if (eptr()) epptr()} != \text{rhs.a.epptr()} \]
(11.12) \[ \text{getloc()} == \text{rhs.p.getloc()} \]
(11.13) \[ \text{rhs} \text{ is empty but usable, as if std::move(rhs).str()} \text{ was called.} \]

### 29.8.2.2 Assignment and swap

```cpp
basic_stringbuf& operator=(basic_stringbuf&& rhs);
```

**Effects:** After the move assignment `*this` has the observable state it would have had if it had been move constructed from `rhs` (see 29.8.2.1).

**Returns:** `*this`.

```cpp
void swap(basic_stringbuf& rhs) noexcept(see below);
```

**Preconditions:** `allocator_traits<Allocator>::propagate_on_container_swap::value` is true or `get_allocator() == s.get_allocator()` is true.

**Effects:** Exchanges the state of `*this` and `rhs`.

**Remarks:** The expression inside `noexcept` is equivalent to:

```
allocator_traits<Allocator>::propagate_on_container_swap::value ||
allocator_traits<Allocator>::is_always_equal::value.
```

```cpp
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)));
```

**Effects:** Equivalent to: `x.swap(y)`.

### 29.8.2.3 Member functions

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`).

```cpp
void init_buf_ptrs(); // exposition only
```

**Effects:** Initializes the input and output sequences from `buf` according to `mode`.

**Postconditions:**

(3.1) \[ \text{If ios_base::out} \text{ is set in mode, pbase()} \text{ points to buf.front()} \text{ and eptr()} >= pbase() + buf.size()} \text{ is true;} \]
(3.1.1) \[ \text{in addition, if ios_base::ate} \text{ is set in mode, pptr()} == pbase() + buf.size()} \text{ is true,} \]
(3.1.2) \[ \text{otherwise pptr()} == pbase()} \text{ is true.} \]
(3.2) \[ \text{If ios_base::in} \text{ is set in mode, eback()} \text{ points to buf.front()}, \text{ and pp()} eback()} &&
         egptr()} == eback()} + buf.size()} \text{ is true.} \]

**[Note:** 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**]
allocator_type get_allocator() const noexcept;

Returns: buf.get_allocator().

basic_string<charT, traits, Allocator> str() const &;

Effects: Equivalent to:

return basic_string<charT, traits, Allocator>(view(), get_allocator());

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() &&;

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().

Postconditions: The underlying character sequence buf is empty and pbase(), pptr(), eptr(), eback(), gptr(), and egptr() are initialized as if by calling init_buf_ptrs() with an empty buf.

basic_string_view<charT, traits> view() const noexcept;

Let sv be basic_string_view<charT, traits>.

Returns:

— If ios_base::out is set in mode, then sv(pbase(), high_mark-pbase()) is returned.

— Otherwise, if ios_base::in is set in mode, then sv(eback(), egptr()-eback()) is returned.

— Otherwise, sv is returned.

[Note: 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();

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:

buf = std::move(s);
init_buf_ptrs();

29.8.2.4 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:
— If \( \text{traits}::\text{eq} \text{int_type}(c, \text{traits}::\text{eof}()) \) returns \text{false} and if the input sequence has a putback position available, and if \( \text{traits}::\text{eq} \text{to_char_type}(c, \text{gptr}()[-1]) \) returns \text{true}, assigns \( \text{gptr}() - 1 \) to \( \text{gptr}() \).

Returns: \( c \).

— If \( \text{traits}::\text{eq} \text{int_type}(c, \text{traits}::\text{eof}()) \) returns \text{false} and if the input sequence has a putback position available, and if \( \text{mode} \& \text{ios}::\text{out} \) is nonzero, assigns \( c \) to \( \ast\ast\text{gptr}() \).

Returns: \( c \).

— If \( \text{traits}::\text{eq} \text{int_type}(c, \text{traits}::\text{eof}()) \) returns \text{true} and if the input sequence has a putback position available, assigns \( \text{gptr}() - 1 \) to \( \text{gptr}() \).

Returns: \( \text{traits}::\text{not\_eof}(c) \).

\( \text{int} \_\text{type} \text{overflow} \text{int} \_\text{type} \ c = \text{traits}::\text{eof}() \) \text{ override;}

\text{Effects}: Appends the character designated by \( c \) to the output sequence, if possible, in one of two ways:

— If \( \text{traits}::\text{eq} \text{int_type}(c, \text{traits}::\text{eof}()) \) returns \text{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 \( \text{sputc}(c) \).

Signals success by returning \( c \).

— If \( \text{traits}::\text{eq} \text{int_type}(c, \text{traits}::\text{eof}()) \) returns \text{true}, there is no character to append.

Signals success by returning a value other than \( \text{traits}::\text{eof}() \).

\text{Remarks}: The function can alter the number of write positions available as a result of any call.

Returns: As specified above, or \( \text{traits}::\text{eof}() \) to indicate failure.

The function can make a write position available only if \( \text{ios}::\text{out} \) is set in \( \text{mode} \). 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 \( \text{ios}::\text{in} \) is set in \( \text{mode} \), the function alters the read end pointer \( \text{egptr}() \) to point just past the new write position.

\( \text{pos} \_\text{type} \text{seekoff} \text{off} \_\text{type} \ \text{off} \text{,} \ \text{ios}::\text{seekdir} \ \text{way}, \)
\( \text{ios}::\text{openmode} \ \text{which} \)
\( = \text{ios}::\text{in} \mid \text{ios}::\text{out} \) \text{ override;}

\text{Effects}: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 124.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ios}::\text{in} ) is set in which</td>
<td>positions the input sequence</td>
</tr>
<tr>
<td>( \text{ios}::\text{out} ) is set in which</td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>both ( \text{ios}::\text{in} ) and ( \text{ios}::\text{out} ) are set in which and either way ( \text{=} ) ( \text{ios}::\text{beg} ) or way ( \text{=} ) ( \text{ios}::\text{end} )</td>
<td>positions both the input and the output sequences</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

For a sequence to be positioned, the function determines \( \text{newoff} \) as indicated in Table 125. If the sequence’s next pointer (either \( \text{gptr}() \) or \( \text{pptr}() \)) is a null pointer and \( \text{newoff} \) is nonzero, the positioning operation fails.

If \( \text{newoff} \lt 0 \), or if \( \text{newoff} \lt 0 \) refers to an uninitialized character \( \text{29.8.2.3} \), the positioning operation fails. Otherwise, the function assigns \( \text{xbeg} + \text{newoff} + \text{off} \) to the next pointer \( \text{xnext} \).
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>the next pointer minus the beginning pointer (<code>xnext - xbeg</code>).</td>
</tr>
<tr>
<td><code>way == ios_base::end</code></td>
<td>the high mark pointer minus the beginning pointer (<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);`  
17 Effects: implementation-defined, except that `setbuf(0, 0)` has no effect.
18 Returns: this.

29.8.3 Class template basic_istringstream [iostreams]

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;

            // 29.8.3.1, 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);  
            explicit basic_istringstream(  
                basic_string<charT, traits, Allocator>&& s,  
                ios_base::openmode which = ios_base::in);  
            template<class SAlloc>  
                basic_istringstream(  
                    const basic_string<charT, traits, SAlloc>& s, const Allocator& a)  
                    : basic_istringstream(s, ios_base::in, a) {}  
            template<class SAlloc>  
                basic_istringstream(  
                    const basic_string<charT, traits, SAlloc>& s,  
                    ios_base::openmode which, const Allocator& a);  
            template<class SAlloc>  
                explicit basic_istringstream(  
                    const basic_string<charT, traits, SAlloc>& s,  
                    ios_base::openmode which = ios_base::in);  

    } // class basic_istringstream
basic_istringstream(const basicistringstream&) = delete;
basic_istringstream(basicistringstream&& rhs);

// 29.8.3.2. assign and swap
basic_istringstream& operator=(const basicistringstream&) = delete;
basic_istringstream& operator=(basicistringstream&& rhs);
void swap(basicistringstream& rhs);

// 29.8.3.3. 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_istringstream<charT, traits, Allocator>& x,
    basic_istringstream<charT, traits, Allocator>& y);

1 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:

— sb, the stringbuf object.

29.8.3.1  Constructors

explicit basic_istringstream(ios_base::openmode which);
1 Effects: Initializes the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1) and sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::in) (29.8.2.1).

explicit basic_istringstream(
    const basic_string<charT, traits, Allocator>& s,
    ios_base::openmode which = ios_base::in);
2 Effects: Initializes the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::in) (29.8.2.1).

basic_istringstream(ios_base::openmode which, const Allocator& a);
3 Effects: Initializes the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1) and sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::in, a) (29.8.2.1).

explicit basic_istringstream(
    basic_string<charT, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::in);
4 Effects: Initializes the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1) and sb with basic_stringbuf<charT, traits, Allocator>(std::move(s), which | ios_base::in) (29.8.2.1).

template<class SAlloc>
basic_istringstream(
    const basic_string<charT, traits, SAlloc>& s,
ios_base::openmode which, const Allocator& a);

5 Effects: Initializes the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::in, a) (29.8.2.1).

template<class SAlloc>
explicit basic_istringstream(basic_istringstream&& rhs);

6 Effects: Initializes the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::in) (29.8.2.1).

7 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.

29.8.3.2 Assignment and swap [istringstream.assign]

void swap(basic_istringstream& rhs);

1 Effects: Equivalent to:

   basic_istream<charT, 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);

2 Effects: Equivalent to: x.swap(y).

29.8.3.3 Member functions [istringstream.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;

3 Effects: Equivalent to: return rdbuf()->str(sa);

basic_string<charT, traits, Allocator> str() &&;

4 Effects: Equivalent to: return std::move(*rdbuf()).str();

basic_string_view<charT, traits> view() const nocexcept;

5 Effects: Equivalent to: return rdbuf()->view();

void str(const basic_string<charT, traits, Allocator>& s);

6 Effects: Equivalent to: rdbuf()->str(s);

template<class SAlloc>
void str(const basic_string<charT, traits, SAlloc>& s);

7 Effects: Equivalent to: rdbuf()->str(s);

void str(basic_string<charT, traits, Allocator>&& s);

8 Effects: Equivalent to: rdbuf()->str(std::move(s));
29.8.4 Class template basic_ostringstream

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;

    // 29.8.4.1, 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);
    explicit basic_ostringstream(basic_string<charT, traits, Allocator>&& s,
                                 ios_base::openmode which = ios_base::out);
    template<class SAlloc>
    basic_ostringstream(const basic_string<charT, traits, Allocator>& s,
                         const Allocator& a);
    template<class SAlloc>
    basic_ostringstream(const basic_string<charT, traits, Allocator>& s,
                         ios_base::openmode which, const Allocator& a);
    explicit basic_ostringstream(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, Allocator>& s,
                         SAlloc& sa);
    template<class SAlloc>
    basic_ostringstream(const basic_string<charT, traits, Allocator>& s,
                         ios_base::openmode which = ios_base::out);
    basic_ostringstream(const basic_ostringstream&) = delete;
    basic_ostringstream(basic_ostringstream&& rhs);
    basic_ostringstream& operator=(const basic_ostringstream&) = delete;
    basic_ostringstream& operator=(basic_ostringstream&& rhs);
    void swap(basic_ostringstream& rhs);

    // 29.8.4.2, assign and swap
    basic_ostringstream& operator==(const basic_ostringstream&);
    basic_ostringstream& operator==(basic_ostringstream& rhs);
    void swap(basic_ostringstream& rhs);

    // 29.8.4.3, 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
    };

§ 29.8.4
template<class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
    basic_ostringstream<charT, traits, Allocator>& y);
}

The class basic_ostringstream<charT, traits, Allocator> supports writing objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— sb, the stringbuf object.

29.8.4.1 Constructors

```c++
explicit basic_ostringstream(ios_base::openmode which);

Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1) and sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::out) (29.8.2.1).
```

```c++
explicit basic_ostringstream(
    const basic_string<charT, traits, Allocator>& s,
    ios_base::openmode which = ios_base::out);

Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::out) (29.8.2.1).
```

```c++
basic_ostringstream(ios_base::openmode which, const Allocator& a);

Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1) and sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::out, a) (29.8.2.1).
```

```c++
explicit basic_ostringstream(
    basic_string<charT, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::out);

Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1) and sb with basic_stringbuf<charT, traits, Allocator>(std::move(s), which | ios_base::out) (29.8.2.1).
```

```c++
template<class SAlloc>

basic_ostringstream(
    const basic_string<charT, traits, SAlloc>& s,
    ios_base::openmode which, const Allocator& a);

Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::out, a) (29.8.2.1).
```

```c++
template<class SAlloc>

explicit basic_ostringstream(
    const basic_string<charT, traits, SAlloc>& s,
    ios_base::openmode which = ios_base::out);

Constraints: is_same_v<SAlloc, Allocator> is false.

Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which | ios_base::out) (29.8.2.1).
```

```c++

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.
29.8.4.2 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).

29.8.4.3 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 SAlloc>

basic_string<charT,traits,SAlloc> str(const SAlloc& sa) const;

Effects: Equivalent to: return rdbuf()->str(sa);

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: 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

namespace std {

template<class charT, class traits = char_traits<charT>,
class Allocator = allocator<charT>>

class basic_stringstream : public basic_iosstring<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, constructors

basic_stringstream() : basic_stringstream(ios_base::out | ios_base::in) {};

explicit basic_stringstream(ios_base::openmode which);

explicit basic_stringstream(
    const basic_string<charT, traits, Allocator>& s,
    ios_base::openmode which = ios_base::out | ios_base::in);

basic_stringstream(ios_base::openmode which, const Allocator& a);

};
explicit basic_stringstream(
    basic_string<charT, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::out | ios_base::in);

template<class SAlloc>
    basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        const Allocator& a)
    : basic_stringstream(s, ios_base::out | ios_base::in, a) {}

template<class SAlloc>
    basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which, const Allocator& a);

    explicit basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which = ios_base::out | ios_base::in);

    basic_stringstream(const basic_stringstream&);  // 29.8.5.2, assign and swap
    basic_stringstream& operator=(const basic_stringstream&);  // 29.8.5.2, assign and swap
    basic_stringstream& operator=(basic_stringstream&& rhs);

    void swap(basic_stringstream& rhs);

    // 29.8.5.3, 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> sb;  // exposition only
    
    template<class charT, class traits, class Allocator>
        void swap(basic_stringstream<charT, traits, Allocator>& x,
            basic_stringstream<charT, traits, Allocator>&& y);  

1 The class template basic_stringstream supports reading and writing from objects of class basic_string<charT, traits, Allocator>. It uses a basic_stringbuf object to control the associated sequence. For the sake of exposition, the maintained data is presented here as — sb, the stringbuf object.

29.8.5.1 Constructors

explicit basic_stringstream(ios_base::openmode which);

1 Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) 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)) (29.7.4.6.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which).
basic_stringstream(ios_base::openmode which, const Allocator& a);

Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) and sb with basic_stringbuf<charT, traits, Allocator>(which, a) (29.8.2.1).

explicit basic_stringstream(
    basic_string<charT, traits, Allocator>&& s,
    ios_base::openmode which = ios_base::out | ios_base::in);

Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) and sb with basic_stringbuf<charT, traits, Allocator>(std::move(s), which) (29.8.2.1).

template<class SAlloc>
    basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which, const Allocator& a);

Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which, a) (29.8.2.1).

template<class SAlloc>
    explicit basic_stringstream(
        const basic_string<charT, traits, SAlloc>& s,
        ios_base::openmode which = ios_base::out | ios_base::in);

Constraints: is_same_v<SAlloc,Allocator> is false.

Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) and sb with basic_stringbuf<charT, traits, Allocator>(s, which) (29.8.2.1).

basic_stringstream(basic_stringstream&& 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<charT, traits>::set_rdbuf(addressof(sb)) to install the contained basic_stringbuf.

### 29.8.5.2 Assignment and swap [stringstream.assign]

void swap(basic_stringstream& rhs);

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);

Effects: Equivalent to: x.swap(y).

### 29.8.5.3 Member functions [stringstream.members]

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()-&gt;str();

template<class SAlloc>
    basic_string<charT,traits,SAlloc> str(const SAlloc& sa) const;

Effects: Equivalent to: return rdbuf()-&gt;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()-&gt;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: rdbuf()->str(s);

void str(basic_string<charT, traits, Allocator>&& s);

Effects: Equivalent to: rdbuf()->str(std::move(s));

29.9 File-based streams

29.9.1 Header <fstream> synopsis

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_filebuf;
    using filebuf = basic_filebuf<char>;
    using wfilebuf = basic_filebuf<wchar_t>;

    template<class charT, class traits = char_traits<charT>>
    class basic_ifstream;
    using ifstream = basic_ifstream<char>;
    using wifstream = basic_ifstream<wchar_t>;

    template<class charT, class traits = char_traits<charT>>
    class basic_ofstream;
    using ofstream = basic_ofstream<char>;
    using wofstream = basic_ofstream<wchar_t>;

    template<class charT, class traits = char_traits<charT>>
    class basic_fstream;
    using fstream = basic_fstream<char>;
    using wfstream = basic_fstream<wchar_t>;
}

The header <fstream> defines four class templates and eight types that associate stream buffers with files and assist reading and writing files.

[Note: The class template basic_filebuf treats a file as a source or sink of bytes. In an environment that uses a large character set, the file typically holds multibyte character sequences and the basic_filebuf object converts those multibyte sequences into wide character sequences. —end note]

In subclause 29.9, member functions taking arguments of const filesystem::path::value_type* are only be provided on systems where filesystem::path::value_type (29.11.7) is not char. [Note: These functions enable class path support for systems with a wide native path character type, such as wchar_t. —end note]

29.9.2 Class template basic_filebuf

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_filebuf : 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;

            // 29.9.2.1, constructors/destructor
            basic_filebuf();
            basic_filebuf(const basic_filebuf&) = delete;
            basic_filebuf(basic_filebuf&& rhs);
            virtual ~basic_filebuf();

§ 29.9.2 1421
// 29.9.2.2, assign and swap
basic_filebuf& operator=(const basic_filebuf&) = delete;
basic_filebuf& operator=(basic_filebuf&& rhs);
void swap(basic_filebuf& rhs);

// 29.9.2.3, members
bool is_open() const;
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
basic_filebuf* open(const string& s,
    ios_base::openmode mode);
basic_filebuf* open(const filesystem::path& s,
    ios_base::openmode mode);
basic_filebuf* close();

protected:
// 29.9.2.4, overridden virtual functions
streamsize showmanyc() override;
int_type underflow() override;
int_type uflow() override;
int_type pbackfail(int_type c = traits::eof()) override;
int_type overflow (int_type c = traits::eof()) override;
basic_streambuf<charT, traits>* setbuf(char_type* s,
    streamsize n) 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;

int sync() override;
void imbue(const locale& loc) override;

};

template<class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
    basic_filebuf<charT, traits>& y);

1 The class basic_filebuf<charT, traits> associates both the input sequence and the output sequence with a file.

2 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.

3 In particular:
   (3.1) — If the file is not open for reading the input sequence cannot be read.
   (3.2) — If the file is not open for writing the output sequence cannot be written.
   (3.3) — A joint file position is maintained for both the input sequence and the output sequence.

4 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.

5 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
   const codecvt<charT, char, typename traits::state_type>& a_codecvt =
   use_facet<codecvt<charT, char, typename traits::state_type>>(getloc());

29.9.2.1 Constructors

basic_filebuf();

1 Effects: Initializes the base class with basic_streambuf<charT, traits> () (29.6.3.1).
2 Postconditions: is_open() == false.

basic_filebuf(basic_filebuf&& rhs);

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.

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.

(4.1) is_open() == rhs_p.is_open()
(4.2) rhs_a.is_open() == false
(4.3) gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
(4.4) egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
(4.5) pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()
(4.6) eptr() - pbase() == rhs_p.eptr() - rhs_p.pbase()
(4.7) if (eback()) eback() != rhs_a.eback()
(4.8) if (gptr()) gptr() != rhs_a.gptr()
(4.9) if (egptr()) egptr() != rhs_a.egptr()
(4.10) if (pbase()) pbase() != rhs_a.pbase()
(4.11) if (pptr()) pptr() != rhs_a.pptr()
(4.12) if (eptr()) eptr() != rhs_a.eptr()

virtual ~basic_filebuf();

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 16.5.5.13).

29.9.2.2 Assignment and swap [filebuf.assign]

basic_filebuf& operator=(basic_filebuf&& rhs);

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 29.9.2.1).

Returns: *this.

void swap(basic_filebuf& rhs);

Effects: Exchanges the state of *this and rhs.

template<class charT, class traits>
void swap(basic_filebuf< charT, traits>& x,
        basic_filebuf< charT, traits>& y);

Effects: Equivalent to: x.swap(y).

29.9.2.3 Member functions [filebuf.members]

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 (16.3.18).

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 [tab:filebuf.open.modes]

<table>
<thead>
<tr>
<th>ios_base flag combination</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary in out trunc app</td>
<td></td>
</tr>
<tr>
<td>+ &quot;w&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + &quot;a&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + &quot;a&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + &quot;y&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + &quot;r&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + &quot;r+&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + &quot;w+&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + &quot;a+&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + &quot;a+b&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + &quot;wb&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;wb&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;ab&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;ab&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;ab&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;ab&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;ab&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;rb&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + &quot;r+b&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + + &quot;w+b&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + + &quot;a+b&quot;</td>
<td></td>
</tr>
<tr>
<td>+ + + + + + &quot;a+b&quot;</td>
<td></td>
</tr>
</tbody>
</table>

4 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`).\(^{323}\)

5 If the repositioning operation fails, calls `close()` and returns a null pointer to indicate failure.

6 Returns: `this` if successful, a null pointer otherwise.

basic_filebuf* open(const string& s, ios_base::openmode mode);

7 basic_filebuf* open(const filesystem::path& s, ios_base::openmode mode);

6 Returns: `open(s.c_str(), mode);

basic_filebuf* close();

8 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, 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.

9 Returns: `this` on success, a null pointer otherwise.

Postconditions: `is_open()` == false.

29.9.2.4 Overridden virtual functions [filebuf.virtuals]

streamsize showmanyc() override;

1 Effects: Behaves the same as `basic_streambuf::showmanyc()` (29.6.3.4).

2 Remarks: An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence.

\(^{323}\) 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).
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:

```c
char extern_buf[XSIZE];
char* extern_end;
charT intern_buf[ISIZE];
charT* intern_end;
```

cdvct_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 with the same method as used by underflow.

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:

- 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.

- 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.

- 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();
charT* end;
char xbuf[XSIZE];
char* xbuf_end;
```

cdvct_base::result r =
   a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);

and then

- If r == cdvct_base::error then fail.
- If r == cdvct_base::noconv then output characters from b up to (and not including) p.
— If \( r == \text{codecvt	extunderscore base	extunderscore partial} \) then output to the file characters from \( \text{xbuf} \) up to \( \text{xbuf	extunderscore end} \), and repeat using characters from \( \text{end} \) to \( \text{p} \). If output fails, fail (without repeating).

— Otherwise output from \( \text{xbuf} \) to \( \text{xbuf	extunderscore end} \), and fail if output fails. At this point if \( \text{b} != \text{p} \) and \( \text{b} == \text{end} \) (\( \text{xbuf} \) isn’t large enough) then increase \( \text{XSIZE} \) and repeat from the beginning.

Returns: \( \text{traits::not	extunderscore eof(c)} \) to indicate success, and \( \text{traits::eof()} \) to indicate failure. If \( \text{is	extunderscore open()} == \text{false} \), the function always fails.

\[
\text{basic	extunderscore streambuf} \ast \text{setbuf(char	extunderscore type} \ast \text{s, streamsize} \ast \text{n) override;}
\]

Effects: If \( \text{setbuf}(0, 0) \) 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 \( \text{pbase()} \) and \( \text{pptr()} \) always return null and output to the file should appear as soon as possible.

\[
\text{pos	extunderscore type} \text{seekoff(off	extunderscore type} \ast \text{off, ios	extunderscore base}::\text{seekdir} \ast \text{way,}
\]

\[
\text{ios	extunderscore base}::\text{openmode which}
\]

\[
= \text{ios	extunderscore base}::\text{in} \mid \text{ios	extunderscore base}::\text{out} \text{ override;}
\]

Effects: Let \( \text{width} \) denote \( \text{a	extunderscore codecvt.encoding()} \). If \( \text{is	extunderscore open()} == \text{false} \), or \( \text{off} != 0 \) \&\& \( \text{width} <= 0 \), then the positioning operation fails. Otherwise, if \( \text{way} != \text{basic	extunderscore ios}::\text{cur} \) or \( \text{off} != 0 \), and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if \( \text{width} > 0 \), call \( \text{fseek(file, width \ast off, whence)} \), otherwise call \( \text{fseek(file, 0, whence)} \).

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 \( \text{width} \) is less than zero then call \( \text{a	extunderscore codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf	extunderscore end)} \) and output the resulting unshift sequence. The function determines one of three values for the argument \( \text{whence} \), of type \( \text{int} \), as indicated in Table 127.

Table 127: \( \text{seekoff} \) effects [tab:filebuf.seekoff]

<table>
<thead>
<tr>
<th>way</th>
<th>stdio Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{basic	extunderscore ios}::beg</td>
<td>SEEK_SET</td>
</tr>
<tr>
<td>\text{basic	extunderscore ios}::cur</td>
<td>SEEK_CUR</td>
</tr>
<tr>
<td>\text{basic	extunderscore ios}::end</td>
<td>SEEK_END</td>
</tr>
</tbody>
</table>

Returns: A newly constructed \( \text{pos	extunderscore type} \) 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 \( \text{pos	extunderscore type}(\text{off	extunderscore type}(-1)) \).

\[
\text{pos	extunderscore type} \text{seekpos(pos	extunderscore type} \ast \text{sp,}
\]

\[
\text{ios	extunderscore base}::\text{openmode which}
\]

\[
= \text{ios	extunderscore base}::\text{in} \mid \text{ios	extunderscore base}::\text{out} \text{ override;}
\]

Alters the file position, if possible, to correspond to the position stored in \( \text{sp} \) (as described below). Altering the file position performs as follows:

1. if \( \text{(om} \& \text{ios	extunderscore base}::\text{out}) != 0 \), then update the output sequence and write any unshift sequence;
2. set the file position to \( \text{sp} \) as if by a call to \( \text{fsetpos} \);
3. if \( \text{(om} \& \text{ios	extunderscore base}::\text{in}) != 0 \), then update the input sequence;

where \( \text{om} \) is the open mode passed to the last call to \( \text{open()} \). The operation fails if \( \text{is	extunderscore open()} \) returns \( \text{false} \).

If \( \text{sp} \) is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If \( \text{sp} \) has not been obtained by a previous successful call to one of the positioning functions (\( \text{seekoff} \) or \( \text{seekpos} \)) on the same file the effects are undefined.

Returns: \( \text{sp} \) on success. Otherwise returns \( \text{pos	extunderscore type}(\text{off	extunderscore type}(-1)) \).

\[
\text{int sync()} \text{ override;}
\]

Effects: If a put area exists, calls \( \text{filebuf}::\text{overflow} \) to write the characters to the file, then flushes the file as if by calling \( \text{fflush(file)} \). If a get area exists, the effect is implementation-defined.
void imbue(const locale& loc) override;

20 Preconditions: If the file is not positioned at its beginning and the encoding of the current locale as determined by a_codecvt.encoding() is state-dependent (28.4.1.4.2) then that facet is the same as the corresponding facet of loc.

21 Effects: Causes characters inserted or extracted after this call to be converted according to loc until another call of imbue.

22 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

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;

    // 29.9.3.1, 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);

    // 29.9.3.2, assign and swap
    basic_ifstream& operator=(const basic_ifstream&) = delete;
    basic_ifstream& operator=(basic_ifstream&& rhs);
    void swap(basic_ifstream& rhs);

    // 29.9.3.3, members
    basic_filebuf<charT, traits>* rdbuf() const;
    bool is_open() const;
    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
    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);
    void close();
  private:
    basic_filebuf<charT, traits> sb; // exposition only
  };
}

1 The class basic_ifstream<charT, traits> supports reading from 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:

(1.1) — sb, the filebuf object.
29.9.3.1 Constructors [ifstream.cons]

basic_ifstream();

Effects: Initializes the base class with basic_istream<charT, traits> (29.7.4.1.1) and sb with basic_filebuf<charT, traits>() (29.9.2.1).

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

Effects: Initializes the base class with basic_istream<charT, traits> (29.7.4.1.1) and sb with basic_filebuf<charT, traits>() (29.9.2.1), then calls rdbuf()->open(s, mode | ios_base::in). If that function returns a null pointer, calls setstate(failbit).

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);

Effects: Equivalent to: basic_ifstream(s.c_str(), mode).

basic_ifstream(basic_ifstream&& rhs);

Effects: Move constructs the base class, and the contained basic_filebuf. Then calls basic_istream<charT, traits>::set_rdbuf(addressof(sb)) to install the contained basic_filebuf.

29.9.3.2 Assignment and swap [ifstream.assign]

void swap(basic_ifstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_istream<charT, traits>::swap(rhs) and sb.swap(rhs.sb).

template<class charT, class traits>
void swap(basic_ifstream<charT, traits>& x,
basic_ifstream<charT, traits>& y);

Effects: Equivalent to: x.swap(y).

29.9.3.3 Member functions [ifstream.members]

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).

§ 29.9.3.3
### 29.9.4 Class template basic_ofstream

```cpp
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;

    // 29.9.4.1, 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);
    basic_ofstream(const basic_ofstream&) = delete;
    basic_ofstream(basic_ofstream&& rhs);

    // 29.9.4.2, assign and swap
    basic_ofstream& operator=(const basic_ofstream&) = delete;
    basic_ofstream& operator=(basic_ofstream&& rhs);
    void swap(basic_ofstream& rhs);

    // 29.9.4.3, 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,
              // wide systems only; see 29.9.1
              ios_base::openmode mode = ios_base::out);
    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);
    void close();
private:
    basic_filebuf<charT, traits> sb;
};
```

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:

1. `sb`, the filebuf object.

#### 29.9.4.1 Constructors

```cpp
basic_ofstream();
```

**Effects:** Initializes the base class with `basic_ostream<charT, traits>(addressof(sb))` (29.7.5.1.1) and `sb` with `basic_filebuf<charT, traits>()` (29.9.2.1).
explicit basic_ofstream(const filesystem::path::value_type* s,
    ios_base::openmode mode = ios_base::out); // wide systems only; see 29.9.1

  Effects: Initializes the base class with basic_ostream<charT, traits>(addressof(sb))
           (29.7.5.1.1) and sb with basic_filebuf<charT, traits>() (29.9.2.1),
           then calls rdbuf()->open(s, mode | ios_base::out). If that function returns a null
           pointer, calls setstate(failbit).

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);

  Effects: Equivalent to: basic_ofstream(s.c_str(), mode).

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.

29.9.4.2 Assignment and swap [fstream.assign]
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).

template<class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
          basic_ofstream<charT, traits>& y);

  Effects: Equivalent to: x.swap(y).

29.9.4.3 Member functions [fstream.members]

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::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 [fstream]

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 1430
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:

- `sb`, the `basic_filebuf` object.

### 29.9.5.1 Constructors

```cpp
basic_fstream();
```

**Effects:** Initializes the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and `sb` with `basic_filebuf<charT, traits>()`.

```cpp
explicit basic_fstream(
    const char* s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

```cpp
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
```

```cpp
explicit basic_fstream(
    const string& s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

```cpp
explicit basic_fstream(
    const filesystem::path& s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

```cpp
basic_fstream(const basic_fstream&) = delete;
```

```cpp
basic_fstream(basic_fstream&& rhs);
```

```cpp
basic_fstream& operator=(const basic_fstream&) = delete;
```

```cpp
basic_fstream& operator=(basic_fstream&& rhs);
```

```cpp
void swap(basic_fstream& rhs);
```
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
2  Effects: Initializes the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1)  
    and sb with basic_filebuf<charT, traits>(). Then calls rdbuf()->open(s, mode). If that function  
    returns a null pointer, calls setstate(failbit).

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);
3  Effects: Equivalent to: basic_fstream(s.c_str(), mode).

basic_fstream(basic_fstream&& rhs);
4  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.

29.9.5.2 Assignment and swap

void swap(basic_fstream& rhs);
1  Effects: Exchanges the state of *this and rhs by calling basic_iostream<charT, traits>::swap(rhs)  
    and sb.swap(rhs.sb).

template<class charT, class traits>  
void swap(basic_fstream<charT, traits>& x,  
    basic_fstream<charT, traits>& y);
2  Effects: Equivalent to: x.swap(y).

29.9.5.3 Member functions

basic_filebuf<charT, traits>* rdbuf() const;
1  Returns: const_cast<basic_filebuf<charT, traits>*>(addressof(sb)).

bool is_open() const;
2  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
3  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);
4  Effects: Calls open(s.c_str(), mode).

void close();
5  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

```cpp
#include <ostream>  // see 29.7.2
```

namespace std {
  template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
  class basic_syncbuf;
  using syncbuf = basic_syncbuf<char>;
  using wsyncbuf = basic_syncbuf<wchar_t>;

  template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
  class basic_osyncstream;
  using osyncstream = basic_osyncstream<char>;
  using wosyncstream = basic_osyncstream<wchar_t>;
}

§ 29.10.2.1 1433

1 The header <syncstream> provides a mechanism to synchronize execution agents writing to the same stream.

29.10.2 Class template basic_syncbuf

29.10.2.1 Overview
1 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 [syncstream.syncbuf.cons]

basic_syncbuf(streambuf_type* obuf, const Allocator& allocator);

1 Effects: Sets wrapped to obuf.
2 Remarks: A copy of allocator is used to allocate memory for internal buffers holding the associated output.
3 Throws: Nothing unless an exception is thrown by the construction of a mutex or by memory allocation.
4 Postconditions: get_wrapped() == obuf and get_allocator() == allocator are true.

basic_syncbuf(basic_syncbuf&& other);

5 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.
6 Remarks: This constructor disassociates other from its wrapped stream buffer, ensuring destruction of other produces no output.

~basic_syncbuf();

1 Effects: Calls emit().
2 Throws: Nothing. If an exception is thrown from emit(), the destructor catches and ignores that exception.

29.10.2.3 Assignment and swap [syncstream.syncbuf.assign]

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 Returns: *this.
3 Postconditions:
4 (3.1) rhs.get_wrapped() == nullptr is true.
5 (3.2) this->get Allocator() == rhs.get Allocator() is true when
6 allocator_traits<Allocator>::propagate_on_container_move_assignment::value
7 is true; otherwise, the allocator is unchanged.
8 Remarks: This assignment operator disassociates rhs from its wrapped stream buffer, ensuring destruction of rhs produces no output.

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

bool emit();

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.

Returns: true if all of the following conditions hold; otherwise false:

1. \*wrapped == nullptr is false.
2. All of the characters in the associated output were successfully transferred.
3. The call to \*wrapped->pubsync() (if any) succeeded.

Postconditions: On success, the associated output is empty.

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 (6.9.2.1), where each emit() call synchronizes with subsequent emit() calls in that total order.

Remarks: May call member functions of \*wrapped while holding a lock uniquely associated with \*wrapped.

streambuf_type* get_wrapped() const noexcept;

Returns: \*wrapped.

allocator_type get_allocator() const noexcept;

Returns: A copy of the allocator that was set in the constructor or assignment operator.

void set_emit_on_sync(bool b) noexcept;

Effects: emit_on_sync = b.

29.10.2.5 Overridden virtual functions

int sync() override;

Effects: Records that the wrapped stream buffer is to be flushed. Then, if emit_on_sync is true, calls emit(). [Note: If emit_on_sync is false, the actual flush is delayed until a call to emit(). — end note]

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;

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>;
    }
29.10.3.2, construction and destruction

```cpp
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: The member functions of the provided stream buffer might be called from emit() while a lock is held. Care should be taken to ensure that this does not result in deadlock. —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: 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
}  // characters transferred; cout not flushed
```

— end example]

[Example: 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;

4 Returns: `sb.get_wrapped()`.

5 [Example: 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

1 Subclause 29.11 describes operations on file systems and their components, such as paths, regular files, and directories.

2 A file system is a collection of files and their attributes.

3 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.

4 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 (29.11.7.1) in the given directory. The parent directory of other types of files is a directory containing a directory entry for the file under discussion.

5 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: 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 (29.11.7), a string stored by the file is used to modify the pathname resolution. [Note: 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 [fs.conformance]

1 Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance subclauses take that into account.

29.11.2.1 POSIX conformance [fs.conform.9945]

1 Some behavior is specified by reference to POSIX (29.11.3). How such behavior is actually implemented is unspecified. [Note: 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 29.11.6. [Note: 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: 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]

29.11.2.2 Operating system dependent behavior conformance [fs.conform.os]

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.

29.11.2.3 File system race behavior [fs.race.behavior]

1 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 29.11.2.3 introduce a file system race.

2 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: 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 Normative references [fs.norm.ref]

1 Subclause 29.11 mentions commercially available operating systems for purposes of exposition.324

29.11.4 Requirements [fs.req]

1 Throughout subclause 29.11, char, wchar_t, char8_t, char16_t, and char32_t are collectively called encoded character types.

2 Functions with template parameters named EcharT shall not participate in overload resolution unless EcharT is one of the encoded character types.

3 Template parameters named InputIterator shall meet the Cpp17InputIterator requirements (23.3.5.2) and shall have a value type that is one of the encoded character types.

324) POSIX® is a registered trademark of The IEEE. 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 these products.
[Note: 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).

29.11.4.1 Namespaces and headers

29.11.5 Header `<filesystem>` synopsis

```
#include <compare> // see 17.11.1
namespace std::filesystem {
    // 29.11.7, paths
    class path;
    
    // 29.11.7.7, path non-member functions
    void swap(path& lhs, path& rhs) noexcept;
    size_t hash_value(const path& p) noexcept;
    
    // 29.11.8, filesystem errors
    class filesystem_error;
    
    // 29.11.11, directory entries
    class directory_entry;
    
    // 29.11.12, directory iterators
    class directory_iterator;
    
    // 29.11.12.2, range access for directory iterators
    directory_iterator begin(directory_iterator iter) noexcept;
    directory_iterator end(const directory_iterator&) noexcept;
    
    // 29.11.13, recursive directory iterators
    class recursive_directory_iterator;
    
    // 29.11.13.2, range access for recursive directory iterators
    recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
    recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;
    
    // 29.11.10, 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;
    };
    
    // 29.11.9, 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>;
    
    // 29.11.14, 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;

bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec);

bool 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,
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);

path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);

path relative(const path& p, error_code& ec);
path relative(const path& p, const path& base = current_path());
path relative(const path& p, const path& base, error_code& ec);

bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;

uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec);

void rename(const path& from, const path& to);
void rename(const path& from, const path& to, error_code& ec) noexcept;

void resize_file(const path& p, uintmax_t size);
void resize_file(const path& p, uintmax_t size, error_code& ec) noexcept;
Implementations should ensure that the resolution and range of `file_time_type` reflect the operating system dependent resolution and range of file time values.

29.11.6 Error reporting

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: This supports two common use cases:

1. Uses where file system errors are truly exceptional and indicate a serious failure. Throwing an exception is an appropriate response.
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:

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. Failure to allocate storage is reported by throwing an exception as described in 16.5.5.13.

3. Destructors throw nothing.

3 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.

29.11.7 Class path

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: Class `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]
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 (29.11.7.1). The maximum number of elements in the sequence is operating system dependent (29.11.2.2).

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. [Note: Pathnames “.” and “..” are relative paths. — end note]

A pathname is a character string that represents the name of a path. Pathnames are formatted according to the generic pathname format grammar (29.11.7.1) 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: POSIX specifies the mechanism in section 4.11, Pathname resolution. — end example]

```cpp
namespace std::filesystem {
    class path {
        public:
            using value_type = see below;
            using string_type = basic_string<value_type>;
            static constexpr value_type preferred_separator = see below;

            // 29.11.9.1, enumeration format
            enum format;

            // 29.11.7.4.1, 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);
            ~path();

            // 29.11.7.4.2, 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 InputIterator>
                path& assign(const InputIterator first, InputIterator last);

            // 29.11.7.4.3, appends
            path& operator/=(const path& p);
            template<class Source>
                path& operator/=(const Source& source);
            template<class Source>
                path& append(const Source& source);
            template<class InputIterator>
                path& append(InputIterator first, InputIterator last);
```
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);
template<class Source>
  path& operator+=(EcharT x);
template<class Source>
  path& concat(const Source& x);
template<class InputIterator>
  path& concat(InputIterator first, InputIterator last);

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;

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);

const string_type& native() const noexcept;
const value_type* c_str() const noexcept;
operator string_type() const;
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;

template<class EcharT, class traits = char_traits<EcharT>,
  class Allocator = allocator<EcharT>>
  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;

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;

path root_name() const;
path root_directory() const;
path root_path() const;
path relative_path() const;
path parent_path() const;
path filename() const;
path stem() const;
path extension() const;

// 29.11.7.4.10, 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;

// 29.11.7.4.11, generation
path lexically_normal() const;
path lexically_relative(const path& base) const;
path lexically_proximate(const path& base) const;

// 29.11.7.5, iterators
class iterator;
using const_iterator = iterator;
iterator begin() const;
iterator end() const;

// 29.11.7.6, 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);

value_type is a typedef for the operating system dependent encoded character type used to represent
pathnames.

The value of the preferred_separator member is the operating system dependent preferred-separator
character (29.11.7.1).

[Example: 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]

29.11.7.1 Generic pathname format

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_opt
fallback-separator directory-separator_opt

preferred-separator: 
operating system dependent directory separator character

fallback-separator: 
/, if preferred-separator is not /

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: Some operating systems prohibit the ASCII control characters (0x00 – 0x1F) in filenames. — end example] [Note: For wide portability, users may wish to limit 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 (29.11.7). If there are no operating system dependent root-names, at least one implementation-defined root-name is required. [Note: 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: 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]

6 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: The generic pathname grammar (29.11.7.1) 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: 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.
29.11.7.2 Conversions

29.11.7.2.1 Argument format conversions

[Note: The format conversions described in this subclause are not applied on POSIX-based operating systems because on these systems:

(1.1) — The generic format is acceptable as a native path.
(1.2) — There is no need to distinguish between native format and generic format in function arguments.
(1.3) — Paths for regular files and paths for directories share the same syntax.
— end note]

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 (29.11.7.1) or a pathname in the native format (29.11.7). 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: Some operating systems may 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: 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: A path stores a native format pathname (29.11.7.4.6) and acts as if it also stores a generic format pathname, related as given below. The implementation may 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: If \( q \) is the result of converting any path at all, it is the result of converting \( p \). — end note]

29.11.7.2.2 Type and encoding conversions

[fs.path.type.cvt]

The native encoding of an ordinary character string is the operating system dependent current encoding for pathnames (29.11.7). 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: 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: 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: 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.
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.

29.11.7.3 Requirements

In addition to the requirements (29.11.4), function template parameters named Source shall be one of:

1

— basic_string<EcharT, traits, Allocator>. A function argument const Source& source shall have an effective range [source.begin(), source.end()).

— basic_string_view<EcharT, traits>. A function argument const Source& source shall have an effective range [source.begin(), source.end()).

— 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().

— 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().

2

Functions taking template parameters named Source shall not participate in overload resolution unless
Source denotes a type other than path, and either

— Source is a specialization of basic_string or basic_string_view, or

— the qualified-id iterator_traits<decay_t<Source>>::value_type is valid and denotes a possibly
const encoded character type (13.10.2).

3 [Note: See path conversions (29.11.7.2) 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.

29.11.7.4 Members

29.11.7.4.1 Constructors

path() noexcept;

Postconditions: empty() == true.

path(const path& p);

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 (29.11.7.2.1), converting format if required (29.11.7.2.1). 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 (29.11.7.3) or the range [first, last), with the
encoding converted if required (29.11.7.2). Finds the detected-format of s (29.11.7.2.1) 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 a 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 (29.11.7.2.2) 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 (29.11.7.2.1) and constructs an object of class path for which the pathname in that format is s.

Example: 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 (29.11.7.2.2). The resulting wide string is then converted to an ordinary character pathname string in the current native 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

29.11.7.4.2 Assignments

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.

path& operator=(path&& p) noexcept;

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: A valid implementation is swap(p). — end note]

Returns: *this.

path& operator=(string_type&& source);
path& assign(string_type&& source);

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.

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);

Effects: Let \( s \) be the effective range of \( \text{source} \) (29.11.7.3) or the range \([\text{first}, \text{last})\), with the encoding converted if required (29.11.7.2). Finds the detected-format of \( s \) (29.11.7.2.1) and sets the pathname in that format to \( s \).

Returns: \(*\text{this}.*\)

### 29.11.7.4.3 Appends

The append operations use \( \text{operator/=} \) to denote their semantic effect of appending \( \text{preferred-separator} \) when needed.

```cpp
path& operator/=(const path& p);
```

Effects: If \( \text{p.is_absolute()} \) \( || \) (\( \text{p.has_root_name()} \) \&\& \( \text{p.root_name()} \) \(!= \text{root_name()} \)), then \( \text{operator/=} \= \text{p} \).

Otherwise, modifies \( \ast\text{this} \) as if by these steps:

1. If \( \text{p.has_root_directory()} \), then removes any root directory and relative path from the generic format pathname. Otherwise, if \( \text{!has_root_directory()} \) \&\& \( \text{is_absolute()} \) is true or if \( \text{has_filename()} \) is true, then appends \( \text{path::preferred_separator} \) to the generic format pathname.
2. Then appends the native format pathname of \( \text{p} \), omitting any \( \text{root-name} \) from its generic format pathname, to the native format pathname.

Example: Even if \( //\text{host} \) is interpreted as a \( \text{root-name} \), both of the paths \( \text{path("//host")/"foo"} \) and \( \text{path("//host")/"foo"} \) equal \( //\text{host/foo"} \) (although the former might use backslash as the preferred separator).

Expression examples:

// On POSIX.
```cpp
path("foo") /= path("");  // yields path("foo")
path("foo") /= path("/bar");  // yields path("/bar")
```

// On Windows.
```cpp
path("foo") /= path("");  // yields path("foo\")
path("foo") /= path("/bar");  // yields path("/bar")
path("foo") /= path("c:\bar");  // yields path("c:\bar")
```

Expression examples:

// On POSIX.
```cpp
path("foo") /= path("foo") /="foo")
path("foo") /= path("/bar") /="bar")
```

// On Windows.
```cpp
path("foo") /= path("foo\") /="foo")
path("foo") /= path("/bar") /="bar")
path("foo") /= path("c:\bar") /="c:\bar")
```

— end example

Returns: \(*\text{this}.*\)

### 29.11.7.4.4 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);
```

Effects: Equivalent to: \( \text{return operator/=(path(source))} \);

### 29.11.7.4.4 Concatenation

```cpp
template<class Source>
path& operator+=(const Source& x);
```

Effects: Equivalent to: \( \text{return operator/=(path(first, last))} \);
template<class Source>
    path& concat(const Source& x);

Effects: Appends \texttt{path(x).native()} to the pathname in the native format. [Note: This directly manipulates the value of \texttt{native()} and may not be portable between operating systems. — end note]

Returns: *this.

path& operator+=(value_type x);

Effects: Equivalent to: return *this += basic_string_view(&x, 1);

template<class EcharT>
    path& operator+=(EcharT x);

Effects: Equivalent to: return *this += path(first, last);

29.11.7.4.5 Modifiers [fs.path.modifiers]

void clear() noexcept;

Postconditions: empty() == true.

path& make_preferred();

Effects: Each directory-separator of the pathname in the generic format is converted to preferred-separator.

Returns: *this.

[Example:
    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"
    "foo/bar"

    On an operating system where preferred-separator is a backslash, the output is:
    "foo/bar"
    "foo\bar"

— end example]

path& remove_filename();

Effects: Remove the generic format pathname of \texttt{filename()} from the generic format pathname.

Postconditions: !has_filename().

Returns: *this.

[Example:
    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:
    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:**

1. Any existing `extension()` (29.11.4.9) is removed from the pathname in the generic format, then
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
3. `operator+=`(`replacement`);

**Returns:** *this.

void swap(path& rhs) noexcept;

**Effects:** Swaps the contents (in all formats) of the two paths.

**Complexity:** Constant time.

### 29.11.7.4.6 Native format observers

The string returned by all native format observers is in the native pathname format (29.11.7).

```cpp
const string_type& native() const noexcept;
```

**Returns:** The pathname in the native format.

```cpp
const value_type* c_str() const noexcept;
```

**Effects:** Equivalent to: `return native().c_str();`

```cpp
operator string_type() const;
```

**Returns:** `native()`.

**Remarks:** All memory allocation, including for the return value, shall be performed by `a`. Conversion, if any, is specified by 29.11.7.2.

```cpp
template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
basic_string<EcharT, traits, Allocator>
string(const Allocator& a = Allocator()) const;
```

**Returns:** `native()`.

**Remarks:** Conversion, if any, is performed as specified by 29.11.7.2.

### 29.11.7.4.7 Generic format observers

Generic format observer functions return strings formatted according to the generic pathname format (29.11.7.1). A single slash (`'/`) character is used as the directory-separator.

```cpp
template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
```

**Example:** On an operating system that uses backslash as its preferred-separator,

```cpp
path("foo\bar").generic_string()
returns "foo/bar". — end example]
```
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 29.11.7.2.

std::string generic_string() const;
std::wstring generic_wwstring() 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 29.11.7.2.

29.11.7.4.8 Compare

int compare(const path& p) const noexcept;

Returns:

1. Let rootNameComparison be the result of this->root_name().native().compare(p.root_name().native()). If rootNameComparison is not 0, rootNameComparison.
2. Otherwise, if !this->has_root_directory() and p.has_root_directory(), a value less than 0.
3. Otherwise, if this->has_root_directory() and !p.has_root_directory(), a value greater than 0.
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.
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.
6. Otherwise, 0.

Effects: Equivalent to: return compare(path(s));

29.11.7.4.9 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().

§ 29.11.7.4.9
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.

path extension() const;

Returns: A path whose pathname in the generic format is the suffix of filename() not included in stem().

§ 29.11.7.4.10 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 (29.11.7), otherwise false.

  [Example: 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().

29.11.7.4.11 Generation

path lexically_normal() const;

  Returns: A path whose pathname in the generic format is the normal form (29.11.7.1) of the pathname in the generic format of *this.

  [Example:
    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;

  Returns: *this made relative to base. Does not resolve (29.11.7) symlinks. Does not first normalize (29.11.7.1) *this or base.

  Effects: If:
  (4.1)  — root_name() != base.root_name() is true, or
  (4.2)  — is_absolute() != base.is_absolute() is true, or
  (4.3)  — !has_root_directory() && base.has_root_directory() is true, or
  (4.4)  — any filename in relative_path() or base.relative_path() can be interpreted as a root-name, returns path(). [Note: 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,
    (4.5)  — if a == end() and b == base.end(), returns path("."); otherwise
    (4.6)  — 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
    (4.7)  — if n == 0 and (a == end() || a->empty()), returns path("."); otherwise
    (4.8)  — returns an object of class path that is default-constructed, followed by
    (4.8.1)  — application of operator/(path("..")) n times, and then
application of \texttt{operator/=} for each element in \([a, \text{end()}]\).

\begin{itemize}
  \item \texttt{assert(path("/a/d").lexically_relative("/a/b/c") == ".../d");}
  \item \texttt{assert(path("/a/b/c").lexically_relative("/a/d") == ".../b/c");}
  \item \texttt{assert(path("/a/b/c").lexically_relative("a") == "b/c");}
  \item \texttt{assert(path("/a/b/c").lexically_relative("a/b/c/x/y") == ".../..");}
  \item \texttt{assert(path("/a/b/c").lexically_relative("a/b/c") == ".");}
  \item \texttt{assert(path("/a/b").lexically_relative("c/d") == ".../../a/b");}
\end{itemize}

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 \texttt{path} equality. — end example

\[Note: \text{If symlink following semantics are desired, use the operational function relative().} — end note\]

\begin{itemize}
  \item \texttt{path lexically_normal() to *this, base, or both.} — end note
\end{itemize}

29.11.7.5 \textit{Iterators} [fs.path.itr]

Path iterators iterate over the elements of the pathname in the generic format (29.11.7.1).

A \texttt{path::iterator} is a constant iterator meeting all the requirements of a bidirectional iterator (23.3.5.5) except that, for dereferenceable iterators \texttt{a} and \texttt{b} of type \texttt{path::iterator} with \texttt{a == b}, there is no requirement that \texttt{*a} and \texttt{*b} are bound to the same object. Its \texttt{value_type} is \texttt{path}.

Calling any non-const member function of a \texttt{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:

\begin{itemize}
  \item The root-name element, if present.
  \item The root-directory element, if present. [Note: The generic format is required to ensure lexicographical comparison works correctly. — end note]
  \item Each successive filename element, if present.
  \item An empty element, if a trailing non-root directory-separator is present.
\end{itemize}

The backward traversal order is the reverse of forward traversal.

\begin{itemize}
  \item \texttt{iterator begin() const;}
  \item \texttt{Returns: An iterator for the first present element in the traversal list above. If no elements are present, the end iterator.}
  \item \texttt{iterator end() const;}
  \item \texttt{Returns: The end iterator.}
\end{itemize}

29.11.7.6 \textit{Inserter and extractor} [fs.path.io]

\begin{itemize}
  \item \texttt{template<class charT, class traits> friend basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const path& p);} \[Effects: Equivalent to os \texttt{\textless\textgreater quoted(p.string<charT, traits />());} [Note: The quoted function is described in 29.7.8. — end note]\n  \item \texttt{Returns: os.}
\end{itemize}
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.

29.11.7.7 Non-member functions

void swap(path& lhs, path& rhs) noexcept;

   Effects: Equivalent to lhs.swap(rhs).

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).

friend bool operator==(const path& lhs, const path& rhs) noexcept;

   Returns: lhs.compare(rhs) == 0.

   [Note: Path equality and path equivalence have different semantics.

(4.1) — Equality is determined by the path non-member operator==, which considers the two paths’ lexical representations only. [Example: path("foo") == "bar" is never true. — end example]

(4.2) — Equivalence is determined by the equivalent() non-member function, which determines if two paths resolve (29.11.7) to the same file system entity. [Example: equivalent("foo", "bar") will be true when both paths resolve to the same file. — end example]

Programmers wishing to determine if two paths are “the same” must decide if “the same” means “the same representation” or “resolve to the same actual file”, and choose the appropriate function accordingly. — end note]

friend strong_ordering operator<=>(const path& lhs, const path& rhs) noexcept;

   Returns: lhs.compare(rhs) <=> 0.

friend path operator/(const path& lhs, const path& rhs);

   Effects: Equivalent to: return path(lhs) /= rhs;

29.11.8 Class filesystem_error

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.

29.11.8.1 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);

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);

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);

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 NTBS 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.

29.11.9 Enumerations [fs.enum]

29.11.9.1 Enum path::format [fs.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 [tab:fs.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</td>
</tr>
<tr>
<td></td>
<td>implementation-defined. The implementation may inspect the content of</td>
</tr>
<tr>
<td></td>
<td>the character sequence to determine the format. [Note: For POSIX-based</td>
</tr>
<tr>
<td></td>
<td>systems, native and generic formats are equivalent and the character</td>
</tr>
<tr>
<td></td>
<td>sequence should always be interpreted in the same way. — end note]</td>
</tr>
</tbody>
</table>

29.11.9.2 Enum class file_type [fs.enum.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 [tab:fs.enum.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>
<tr>
<td>not_found</td>
<td>Pseudo-type indicating the file was not found. [Note: The file not being found is not considered an error while determining the type of a file. — end note]</td>
</tr>
<tr>
<td>regular</td>
<td>Regular file</td>
</tr>
<tr>
<td>directory</td>
<td>Directory file</td>
</tr>
<tr>
<td>symlink</td>
<td>Symbolic link file</td>
</tr>
<tr>
<td>block</td>
<td>Block special file</td>
</tr>
<tr>
<td>character</td>
<td>Character special file</td>
</tr>
<tr>
<td>fifo</td>
<td>FIFO or pipe file</td>
</tr>
<tr>
<td>socket</td>
<td>Socket file</td>
</tr>
<tr>
<td>implementation-defined</td>
<td>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</td>
</tr>
<tr>
<td>unknown</td>
<td>The file exists but the type could not be determined</td>
</tr>
</tbody>
</table>

29.11.9.3 Enum class copy_options [fs.enum.copy.opts]

1 The enum class type copy_options is a bitmask type (16.4.2.2.4) 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.

Table 130: Enum class copy_options [tab:fs.enum.copy.opts]

<table>
<thead>
<tr>
<th>Option group controlling copy_file function effects for existing target files</th>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Error; file already exists.</td>
<td></td>
</tr>
<tr>
<td>skip_existing</td>
<td>Do not overwrite existing file, do not report an error.</td>
<td></td>
</tr>
<tr>
<td>overwrite_existing</td>
<td>Overwrite the existing file.</td>
<td></td>
</tr>
<tr>
<td>update_existing</td>
<td>Overwrite the existing file if it is older than the replacement file.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option group controlling copy function effects for sub-directories</th>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Do not copy sub-directories.</td>
<td></td>
</tr>
<tr>
<td>recursive</td>
<td>Recursively copy sub-directories and their contents.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option group controlling copy function effects for symbolic links</th>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Follow symbolic links.</td>
<td></td>
</tr>
<tr>
<td>copy_symlinks</td>
<td>Copy symbolic links as symbolic links rather than copying the files that they point to.</td>
<td></td>
</tr>
<tr>
<td>skip_symlinks</td>
<td>Ignore symbolic links.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option group controlling copy function effects for choosing the form of copying</th>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Copy content.</td>
<td></td>
</tr>
<tr>
<td>directories_only</td>
<td>Copy directory structure only, do not copy non-directory files.</td>
<td></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>
<td></td>
</tr>
<tr>
<td>create_hard_links</td>
<td>Make hard links instead of copies of files.</td>
<td></td>
</tr>
</tbody>
</table>
29.11.9.4  Enum class perms

The enum class type perms is a bitmask type (16.4.2.2.4) that specifies bitmask constants used to identify file permissions, with the meanings listed in Table 131.

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>S_IRWXU</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;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>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;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>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;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>others_read</td>
</tr>
<tr>
<td>all</td>
<td>0777</td>
<td></td>
<td>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></td>
<td>all</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>

29.11.9.5  Enum class perm_options

The enum class type perm_options is a bitmask type (16.4.2.2.4) 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 132: Enum class perm_options

<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>

29.11.9.6  Enum class directory_options

The enum class type directory_options is a bitmask type (16.4.2.2.4) 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 133: Enum class directory_options

<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>

29.11.10 Class file_status

namespace std::filesystem {
    class file_status {
        public:
        // 29.11.10.1, 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;
        // 29.11.10.3, modifiers
        void type(file_type ft) noexcept;
        void permissions(perms prms) noexcept;
        // 29.11.10.2, observers
        file_type type() const noexcept;
        perms permissions() const noexcept;

        friend bool operator==(const file_status& lhs, const file_status& rhs) noexcept
        { return lhs.type() == rhs.type() && lhs.permissions() == rhs.permissions(); }
    };
}

An object of type file_status stores information about the type and permissions of a file.

29.11.10.1 Constructors

explicit file_status(file_type ft, perms prms = perms::unknown) noexcept;

Postconditions: type() == ft and permissions() == prms.

29.11.10.2 Observers

file_type type() 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.

perms permissions() const noexcept;

Returns: The value of permissions() specified by the postconditions of the most recent call to a 
constructor, operator=, or permissions(perms) function.

29.11.10.3 Modifiers

void type(file_type ft) noexcept;

Postconditions: type() == ft.
void permissions(perms prms) noexcept;

Postconditions: permissions() == prms.

29.11.11 Class directory_entry

namespace std::filesystem {
    class directory_entry {
    public:
        // 29.11.11.1, 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&) = default;
        directory_entry& operator=(directory_entry&&) noexcept = default;

        // 29.11.11.2, 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;

        // 29.11.11.3, 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_symblic() const;
        bool is_symblic(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;
    }
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 available and storing the values would allow the implementation to eliminate file system accesses by `directory_entry` observer functions (29.11.14). Such stored file attribute values are said to be cached.

[Note: For purposes of exposition, class `directory_iterator` (29.11.12) 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:

```cpp
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]

### 29.11.11.1 Constructors

**Explicit**

```cpp
explicit directory_entry(const filesystem::path& p);
directory_entry(const filesystem::path& p, error_code& ec);
```

**Effects:** Calls `refresh()` or `refresh(ec)`, respectively.

**Postconditions:** `path()` == `p` if no error occurs, otherwise `path()` == `filesystem::path()`.

**Throws:** As specified in 29.11.6.

### 29.11.11.2 Modifiers

```cpp
void assign(const filesystem::path& p);
void assign(const filesystem::path& p, error_code& ec);
```

**Effects:** Equivalent to `pathobject = p`, then `refresh()` or `refresh(ec)`, respectively. If an error occurs, the values of any cached attributes are unspecified.

**Throws:** As specified in 29.11.6.

```cpp
void replace_filename(const filesystem::path& p);
void replace_filename(const filesystem::path& p, error_code& ec);
```

**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.

**Throws:** As specified in 29.11.6.
void refresh();
void refresh(error_code& ec) noexcept;

Effects: Stores the current values of any cached attributes of the file p resolves to. If an error occurs, an error is reported (29.11.6) and the values of any cached attributes are unspecified.

Throws: As specified in 29.11.6.

[Note: Implementations of directory_iterator (29.11.12) are prohibited from directly or indirectly calling the refresh function since it must access the external file system, and the objective of caching is to avoid unnecessary file system accesses. — end note]

29.11.11.3 Observers [fs.dir.entry.obs]

Unqualified function names in the Returns: elements of the directory_entry observers described below refer to members of the std::filesystem namespace.

const filesystem::path& path() const noexcept;
operator const filesystem::path&() const noexcept;

Returns: pathobject.

bool exists() const;
bool exists(error_code& ec) const noexcept;

Returns: exists(this->status()) or exists(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_block_file() const;
bool is_block_file(error_code& ec) const noexcept;

Returns: is_block_file(this->status()) or is_block_file(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_character_file() const;
bool is_character_file(error_code& ec) const noexcept;

Returns: is_character_file(this->status()) or is_character_file(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_directory() const;
bool is_directory(error_code& ec) const noexcept;

Returns: is_directory(this->status()) or is_directory(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_fifo() const;
bool is_fifo(error_code& ec) const noexcept;

Returns: is_fifo(this->status()) or is_fifo(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_other() const;
bool is_other(error_code& ec) const noexcept;

Returns: is_other(this->status()) or is_other(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_regular_file() const;
bool is_regular_file(error_code& ec) const noexcept;

Returns: is_regular_file(this->status()) or is_regular_file(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_socket() const;
bool is_socket(error_code& ec) const noexcept;

Returns: is_socket(this->status()) or is_socket(this->status(ec)), respectively.

Throws: As specified in 29.11.6.

bool is_symlink() const;
bool is_symlink(error_code& ec) const noexcept;

Returns: is_symlink(this->symlink_status()) or is_symlink(this->symlink_status(ec)), respectively.

Throws: As specified in 29.11.6.

uintmax_t file_size() const;
uintmax_t file_size(error_code& ec) const noexcept;

Returns: If cached, the file size attribute value. Otherwise, file_size(path()) or file_size(path(), ec), respectively.

Throws: As specified in 29.11.6.

uintmax_t hard_link_count() const;
uintmax_t hard_link_count(error_code& ec) const noexcept;

Returns: If cached, the hard link count attribute value. Otherwise, hard_link_count(path()) or hard_link_count(path(), ec), respectively.

Throws: As specified in 29.11.6.

file_time_type last_write_time() const;
file_time_type last_write_time(error_code& ec) const noexcept;

Returns: If cached, the last write time attribute value. Otherwise, last_write_time(path()) or last_write_time(path(), ec), respectively.

Throws: As specified in 29.11.6.

file_status status() const;
file_status status(error_code& ec) const noexcept;

Returns: If cached, the status attribute value. Otherwise, status(path()) or status(path(), ec), respectively.

Throws: As specified in 29.11.6.

file_status symlink_status() const;
file_status symlink_status(error_code& ec) const noexcept;

Returns: If cached, the symlink status attribute value. Otherwise, symlink_status(path()) or symlink_status(path(), ec), respectively.

Throws: As specified in 29.11.6.

bool operator==(const directory_entry& rhs) const noexcept;

Returns: pathobject == rhs.pathobject.

strong_ordering operator<=>(const directory_entry& rhs) const noexcept;

Returns: pathobject <=> rhs.pathobject.

### 29.11.12 Class directory_iterator

An object of type directory_iterator provides an iterator for a sequence of directory_entry elements representing the path and any cached attribute values (29.11.11) for each file in a directory or in an implementation-defined directory-like file type. [Note: For iteration into sub-directories, see class recursive_directory_iterator (29.11.13). — 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&;

// 29.11.12.1, member functions
directory_iterator() noexcept;
directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
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 23.3.5.2, input iterators
});

1 directory_iterator meets the Cpp17InputIterator requirements (23.3.5.2).
2 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.
3 The end iterator is not dereferenceable.
4 Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator.
5 The result of calling the path() member of the directory_entry object obtained by dereferencing a directory_iterator is a reference to a path object composed of the directory argument from which the iterator was constructed with filename of the directory entry appended as if by operator=/
6 Directory iteration shall not yield directory entries for the current (dot) and parent (dot-dot) directories.
7 The order of directory entries obtained by dereferencing successive increments of a directory_iterator is unspecified.
8 Constructors and non-const directory_iterator member functions store the values of any cached attributes (29.11.11) in the directory_entry element returned by operator*(). directory_iterator member functions shall not directly or indirectly call any directory_entry refresh function. [Note: The exact mechanism for storing cached attribute values is not exposed to users. For exposition, class directory_iterator is shown in 29.11.11 as a friend of class directory_entry. — end note]
9 [Note: Programs performing directory iteration may wish to test if the path obtained by dereferencing a directory iterator actually exists. It could be a symbolic link to a non-existent file. Programs recursively walking directory trees for purposes of removing and renaming entries may wish to avoid following symbolic links. — end note]
10 [Note: If a file is removed from or added to a directory after the construction of a 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 readdir_r. — end note]

29.11.12.1 Members

directory_iterator() noexcept;

1 Effects: Constructs the end iterator.
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);

Effects: For the directory that p resolves to, constructs an iterator for the first element in a sequence of directory_entry elements representing the files in the directory, 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.

Throws: As specified in 29.11.6.

[Note: To iterate over the current directory, use directory_iterator(".") rather than directory_iterator(""). — end note]

directory_iterator(const directory_iterator& rhs);
directory_iterator(directory_iterator&& rhs) noexcept;

Postconditions: *this has the original value of rhs.

directory_iterator& operator=(const directory_iterator& rhs);
directory_iterator& operator=(directory_iterator&& rhs) noexcept;

Effects: If *this and rhs are the same object, the member has no effect.

Postconditions: *this has the original value of rhs.

Returns: *this.

directory_iterator& operator++();
directory_iterator& increment(error_code& ec);

Effects: As specified for the prefix increment operation of Input iterators (23.3.5.2).

Returns: *this.

Throws: As specified in 29.11.6.

29.11.12.2 Non-member functions

These functions enable range access for directory_iterator.

directory_iterator begin(directory_iterator iter) noexcept;

Returns: iter.

directory_iterator end(const directory_iterator&) noexcept;

Returns: directory_iterator().

29.11.13 Class recursive_directory_iterator

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&;

            recursive_directory_iterator() noexcept;
            explicit recursive_directory_iterator(const path& p);
            recursive_directory_iterator(const path& p, directory_options options);
            // 29.11.13.1, constructors and destructor
        }
    }
}
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();

// 29.11.13.1, observers
directory_options options() const;
int depth() const;
bool recursion_pending() const;

const directory_entry& operator*() const;
const directory_entry* operator->() const;

// 29.11.13.1, 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 23.3.5.2, 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.

4 [Note: If the directory structure being iterated over contains cycles then the end iterator may be unreachable. — end note]

§ 29.11.13.1 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 29.11.6.

[Note: To iterate over the current directory, use recursive_directory_iterator(".") rather than recursive_directory_iterator(""). — end note]
[Note: By default, `recursive_directory_iterator` does not follow directory symlinks. To follow directory symlinks, specify `options` as `directory_options::follow_directory_symlink`. — end note]

`recursive_directory_iterator(const recursive_directory_iterator& rhs);`

**Postconditions:**

(7.1) \[ \text{options()} == \text{rhs.options()} \]
(7.2) \[ \text{depth()} == \text{rhs.depth()} \]
(7.3) \[ \text{recursion_pending()} == \text{rhs.recursion_pending()} \]

`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) \[ \text{options()} == \text{rhs.options()} \]
(10.2) \[ \text{depth()} == \text{rhs.depth()} \]
(10.3) \[ \text{recursion_pending()} == \text{rhs.recursion_pending()} \]

**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: 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 (23.3.5.2), 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
(\text{options()} \& \text{directory_options::skip_permission_denied}) \neq \text{directory_options::none}

and an error occurs indicating that permission to access directory \((\ast \text{this})\rightarrow\text{path}()) is denied, then directory \((\ast \text{this})\rightarrow\text{path}()) is treated as an empty directory and no error is reported.

\textbf{Returns: } \ast \text{this}.

\textbf{Throws: } As specified in 29.11.6.

\texttt{void pop();}
\texttt{void pop(error_code& ec);}  
\begin{itemize}
\item \textbf{Effects: } If \texttt{depth()} == 0, set \ast \text{this} to \texttt{recursive_directory_iterator().} Otherwise, cease iteration of the directory currently being iterated over, and continue iteration over the parent directory.
\item \textbf{Throws: } As specified in 29.11.6.
\item \textbf{Remarks: } Any copies of the previous value of \ast \text{this} are no longer required to be dereferenceable nor to be in the domain of \==.
\end{itemize}

\texttt{void disable_recursion_pending();}

\textbf{Postconditions: } \texttt{recursion_pending()} == \texttt{false}.

\begin{itemize}
\item \texttt{[Note: disable_recursion_pending()} is used to prevent unwanted recursion into a directory. \texttt{— end note]}
\end{itemize}

\textbf{29.11.13.2 Non-member functions} \hspace{1em} \texttt{[fs.rec.dir.itr.nonmembers]}

These functions enable use of \texttt{recursive_directory_iterator} with range-based for statements.

\texttt{recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;}

\textbf{Returns: } \texttt{iter}.

\texttt{recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;}

\textbf{Returns: } \texttt{recursive_directory_iterator().}

\textbf{29.11.14 Filesystem operation functions} \hspace{1em} \texttt{[fs.opfuncs]}

Filesystem operation functions query or modify files, including directories, in external storage.

\begin{itemize}
\item \texttt{[Note: Because hardware failures, network failures, file system races (29.11.2.3), and many other kinds of errors occur frequently in file system operations, users should be aware that any filesystem operation function, no matter how apparently innocuous, may encounter an error; see 29.11.6. \texttt{— end note]}
\end{itemize}

\textbf{29.11.14.1 Absolute} \hspace{1em} \texttt{[fs.op.absolute]}

\texttt{path absolute(const path& p);}  
\texttt{path absolute(const path& p, error_code& ec);}  
\begin{itemize}
\item \textbf{Effects: } Composes an absolute path referencing the same file system location as \texttt{p} according to the operating system (29.11.2.2).
\item \textbf{Returns: } The composed path. The signature with argument \texttt{ec} returns \texttt{path()} if an error occurs.
\item \texttt{[Note: For the returned path, rp, rp.is_absolute()} is \texttt{true} unless an error occurs. \texttt{— end note]}
\item \textbf{Throws: } As specified in 29.11.6.
\item \texttt{[Note: To resolve symlinks, or perform other sanitization which might require queries to secondary storage, such as hard disks, consider canonical (29.11.14.2). \texttt{— end note]}
\item \texttt{[Note: Implementations are strongly encouraged to not query secondary storage, and not consider !exists(p) an error. \texttt{— end note]}
\item \texttt{[Example: For POSIX-based operating systems, \texttt{absolute(p)} is simply current_path()/p. For}}
\item \texttt{Windows-based operating systems, \texttt{absolute might have the same semantics as GetFullPathNameW.}}
\item \texttt{[end example]}
\end{itemize}

\textbf{29.11.14.2 Canonical} \hspace{1em} \texttt{[fs.op.canonical]}

\texttt{path canonical(const path& p);}
path canonical(const path& p, error_code& ec);

1. **Effects:** Converts `p` to an absolute path that has no symbolic link, dot, or dot-dot elements in its pathname in the generic format.

2. **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.

3. **Throws:** As specified in §29.11.6.

4. **Remarks:** `!exists(p)` is an error.

### 29.11.14.3 Copy

[fs.op.copy]

void copy(const path& from, const path& to);

1. **Effects:** Equivalent to `copy(from, to, copy_options::none)`.

void copy(const path& from, const path& to, error_code& ec);

2. **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);

3. **Preconditions:** At most one element from each option group (§29.11.9.3) is set in `options`.

4. **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 (§29.11.9.2), then the effects are implementation-defined.

   (4.5) Otherwise, an error is reported as specified in §29.11.6 if:

   (4.5.1) `exists(f)` is `false`, or

   (4.5.2) `equivalent(from, to)` is `true`, or

   (4.5.3) `is_other(f) || is_other(t)` is `true`, or

   (4.5.4) `is_directory(f) && is_regular_file(t)` is `true`.

   (4.6) Otherwise, if `is_symlink(f)`, then:

   (4.6.1) If `(options & copy_options::skip_symlinks) != copy_options::none` then return.

   (4.6.2) Otherwise if

   ```
   !exists(t) && (options & copy_options::copy_symlinks) != copy_options::none
   ```

   then `copy_symlink(from, to)`.

   (4.6.3) Otherwise report an error as specified in §29.11.6.

   (4.7) Otherwise, if `is_regular_file(f)`, then:

   (4.7.1) If `(options & copy_options::directories_only) != copy_options::none`, then return.

   (4.7.2) Otherwise, if `(options & copy_options::create_symlinks) != copy_options::none`, then create a symbolic link to the source file.

   (4.7.3) 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)` if
  - `(options & copy_options::create_symlinks) != copy_options::none`

then report an error with an `error_code` argument equal to `make_error_code(errc::is_a_directory)`.

Otherwise, if

- `is_directory(t)` if
  - `((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 29.11.9.3.

Otherwise, for the signature with argument `ec`, `ec.clear()`.

Otherwise, no effects.

**Throws:** As specified in 29.11.6.

**Remarks:** For the signature with argument `ec`, any library functions called by the implementation shall have an `error_code` argument if applicable.

**Example:** Given this directory structure:

```
/dir1
  file1
  file2
  dir2
  file3
```

Calling `copy("/dir1", "/dir3")` would result in:

```
/dir1
  file1
  file2
  dir2
  file3
/dir3
  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]
29.11.14.4 Copy file

bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec);

Returns: copy_file(from, to, copy_options::none) or
          copy_file(from, to, copy_options::none, ec), respectively.

 Throws: As specified in 29.11.6.

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);

Preconditions: At most one element from each option group (29.11.9.3) is set in options.
Effects: As follows:

(4.1) — Report an error as specified in 29.11.6 if:
      — is_regular_file(from) is false, or
      — exists(to) is true and is_regular_file(to) is false, or
      — exists(to) is true and equivalent(from, to) is true, or
      — exists(to) is true and
        (options & (copy_options::skip_existing |
                    copy_options::overwrite_existing |
                    copy_options::update_existing)) == copy_options::none
(4.2) — Otherwise, copy the contents and attributes of the file from resolves to, to the file to resolves to, if:
      — exists(to) is false, or
      — (options & copy_options::overwrite_existing) != copy_options::none, or
      — (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 (29.11.14.25).
(4.3) — Otherwise, no effects.

Returns: true if the from file was copied, otherwise false. The signature with argument ec returns
false if an error occurs.

Throws: As specified in 29.11.6.

Complexity: At most one direct or indirect invocation of status(to).

29.11.14.5 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;

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.

Throws: As specified in 29.11.6.

29.11.14.6 Create directories

bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec);

Effects: Calls create_directory() for each element of p that does not exist.

Returns: true if a new directory was created for the directory p resolves to, otherwise false.

Throws: As specified in 29.11.6.

Complexity: \( \Theta(n) \) where \( n \) is the number of elements of p.
29.11.14.7 Create directory

```cpp
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

**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.

**Returns:** `true` if a new directory was created, otherwise `false`.

**Throws:** As specified in 29.11.6.

```cpp
bool create_directory(const path& p, const path& existing_p);
bool create_directory(const path& p, const path& existing_p, error_code& ec) noexcept;
```

**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. [Note: 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]

**Returns:** `true` if a new directory was created with attributes copied from directory `existing_p`, otherwise `false`.

**Throws:** As specified in 29.11.6.

29.11.14.8 Create directory symlink

```cpp
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;
```

**Effects:** Establishes the postcondition, as if by POSIX `symlink()`.

**Postconditions:** `new_symlink` resolves to a symbolic link file that contains an unspecified representation of `to`.

**Throws:** As specified in 29.11.6.

[Note: Some operating systems require symlink creation to identify that the link is to a directory. Portable code should use `create_directory_symlink()` to create directory symlinks rather than `create_symlink()` —end note]

[Note: 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]

29.11.14.9 Create hard link

```cpp
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;
```

**Effects:** Establishes the postcondition, as if by POSIX `link()`.

**Postconditions:**

1. `exists(to) && exists(new_hard_link) && equivalent(to, new_hard_link)`
2. The contents of the file or directory `to` resolves to are unchanged.

**Throws:** As specified in 29.11.6.

[Note: 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]

29.11.14.10 Create symlink

```cpp
void create_symlink(const path& to, const path& new_symlink);
```
void create_symlink(const path& to, const path& new_symlink, 
    error_code& ec) noexcept;
Effects: Establishes the postcondition, as if by POSIX symlink().
Postconditions: new_symlink resolves to a symbolic link file that contains an unspecified representation 
of to.
Throws: As specified in 29.11.6.
[Note: 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]

29.11.14.11 Current path

path current_path();
path current_path(error_code& ec);
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.
Throws: As specified in 29.11.6.
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.
[Note: The current_path() name was chosen to emphasize that the returned value is a path, not just 
a single directory name. — end note]
[Note: The current path as returned by many operating systems is a dangerous global variable. It may 
be changed unexpectedly by third-party or system library functions, or by another thread. — end 
ote]

void current_path(const path& p);
void current_path(const path& p, error_code& ec) noexcept;
Effects: Establishes the postcondition, as if by POSIX chdir().
Postconditions: equivalent(p, current_path()).
Throws: As specified in 29.11.6.
[Note: The current path for many operating systems is a dangerous global state. It may be changed 
unexpectedly by a third-party or system library functions, or by another thread. — end note]

29.11.14.12 Equivalent

bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2, error_code& ec) noexcept;
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.
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: 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]
Remarks: !exists(p1) || !exists(p2) is an error.
Throws: As specified in 29.11.6.

29.11.14.13 Exists

bool exists(file_status s) noexcept;
Returns: status_known(s) && s.type() != file_type::not_found.
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).
4 Returns: \texttt{exists(s)}.
5 Throws: As specified in 29.11.6.

29.11.14.14 File size

\texttt{uintmax_t file_size(const path& p);}  
\texttt{uintmax_t file_size(const path& p, error_code& ec) noexcept;}  
1 Effects: If \texttt{exists(p)} is false, an error is reported (29.11.6).
2 Returns:  

\begin{enumerate}
\item If \texttt{is_regular_file(p)}, the size in bytes of the file \texttt{p} resolves to, determined as if by the value of the POSIX \texttt{stat} class member \texttt{st_size} obtained as if by POSIX \texttt{stat()}.  
\item Otherwise, the result is implementation-defined.  
\end{enumerate}

The signature with argument \texttt{ec} returns \texttt{static_cast<uintmax_t>(-1)} if an error occurs.
3 Throws: As specified in 29.11.6.

29.11.14.15 Hard link count

\texttt{uintmax_t hard_link_count(const path& p);}  
\texttt{uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;}  
1 Returns: The number of hard links for \texttt{p}. The signature with argument \texttt{ec} returns \texttt{static_cast<uintmax_t>(-1)} if an error occurs.
2 Throws: As specified in 29.11.6.

29.11.14.16 Is block file

\texttt{bool is_block_file(file_status s) noexcept;}  
\texttt{bool is_block_file(const path& p);}  
\texttt{bool is_block_file(const path& p, error_code& ec) noexcept;}  
1 Returns: \texttt{s.type() == file_type::block}.  
2 Returns: \texttt{is_block_file(status(p))} or \texttt{is_block_file(status(p, ec))}, respectively. The signature with argument \texttt{ec} returns \texttt{false} if an error occurs.
3 Throws: As specified in 29.11.6.

29.11.14.17 Is character file

\texttt{bool is_character_file(file_status s) noexcept;}  
\texttt{bool is_character_file(const path& p);}  
\texttt{bool is_character_file(const path& p, error_code& ec) noexcept;}  
1 Returns: \texttt{s.type() == file_type::character}.  
2 Returns: \texttt{is_character_file(status(p))} or \texttt{is_character_file(status(p, ec))}, respectively. The signature with argument \texttt{ec} returns \texttt{false} if an error occurs.
3 Throws: As specified in 29.11.6.

29.11.14.18 Is directory

\texttt{bool is_directory(file_status s) noexcept;}  
\texttt{bool is_directory(const path& p);}  
\texttt{bool is_directory(const path& p, error_code& ec) noexcept;}  
1 Returns: \texttt{s.type() == file_type::directory}.  
2 Returns: \texttt{is_directory(status(p))} or \texttt{is_directory(status(p, ec))}, respectively. The signature with argument \texttt{ec} returns \texttt{false} if an error occurs.
3 Throws: As specified in 29.11.6.
29.11.14.19  Is empty  

bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec);

1  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.

2  Throws: As specified in 29.11.6.

29.11.14.20  Is fifo  

bool is_fifo(file_status s) noexcept;

1  Returns: s.type() == file_type::fifo.

bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;

2  Returns: is_fifo(status(p)) or is_fifo(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

3  Throws: As specified in 29.11.6.

29.11.14.21  Is other  

bool is_other(file_status s) noexcept;

1  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;

2  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 29.11.6.

29.11.14.22  Is regular file  

bool is_regular_file(file_status s) noexcept;

1  Returns: s.type() == file_type::regular.

bool is_regular_file(const path& p);

2  Returns: is_regular_file(status(p)).

3  Throws: filesystem_error if status(p) would throw filesystem_error.

bool is_regular_file(const path& p, error_code& ec) noexcept;

4  Effects: Sets ec as if by status(p, ec). [Note: 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]

5  Returns: is_regular_file(status(p, ec)). Returns false if an error occurs.
### 29.11.14.23 Is socket

```cpp
bool is_socket(file_status s) noexcept;
Returns: s.type() == file_type::socket.
```

```cpp
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.
```

### 29.11.14.24 Is symlink

```cpp
bool is_symlink(file_status s) noexcept;
Returns: s.type() == file_type::symlink.
```

```cpp
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.
```

### 29.11.14.25 Last write time

```cpp
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.
```

```cpp
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().
```

### 29.11.14.26 Permissions

```cpp
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);
Preconditions: Exactly one of the perm_options constants replace, add, or remove is present in opts.
Remarks: The second signature behaves as if it had an additional parameter perm_options opts with an argument of perm_options::replace.
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().
```

### 29.11.14.27 Proximate

```cpp
path proximate(const path& p, error_code& ec);
Returns: proximate(p, current_path(), ec).
```
path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, const path& base, error_code& ec);

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.

Throws: As specified in 29.11.6.

29.11.14.28  Read symlink

path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);

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.

Throws: As specified in 29.11.6. [Note: It is an error if \( p \) does not resolve to a symbolic link. — end note]

29.11.14.29  Relative

path relative(const path& p, error_code& ec);

Returns: relative(p, current_path(), ec).

Throws: As specified in 29.11.6.

path relative(const path& p, const path& base = current_path());
path relative(const path& p, const path& base, error_code& ec);

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.

Throws: As specified in 29.11.6.

29.11.14.30  Remove

bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;

Effects: If exists(symlink_status(p, ec)), the file \( p \) is removed as if by POSIX remove(). [Note: A symbolic link is itself removed, rather than the file it resolves to. — end note]

Postconditions: exists(symlink_status(p)) is false.

Returns: false if \( p \) did not exist, otherwise true. The signature with argument \( ec \) returns false if an error occurs.

Throws: As specified in 29.11.6.

29.11.14.31  Remove all

uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec);

Effects: Recursively deletes the contents of \( p \) if it exists, then deletes file \( p \) itself, as if by POSIX remove(). [Note: A symbolic link is itself removed, rather than the file it resolves to. — end note]

Postconditions: exists(symlink_status(p)) is false.

Returns: The number of files removed. The signature with argument \( ec \) returns static_cast<uintmax_t>(-1) if an error occurs.
Throws: As specified in 29.11.6.

29.11.14.32 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;

Effects: Renames old_p to new_p, as if by POSIX rename().

[Note:
(1.1) — If old_p and new_p resolve to the same existing file, no action is taken.
(1.2) — Otherwise, the rename may 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 may be an error on other operating systems.
A symbolic link is itself renamed, rather than the file it resolves to. — end note]

Throws: As specified in 29.11.6.

29.11.14.33 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;

Effects: Causes the size that would be returned by file_size(p) to be equal to new_size, as if by
POSIX truncate().

Throws: As specified in 29.11.6.

29.11.14.34 Space

space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;

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.

Throws: As specified in 29.11.6.

Remarks: The value of member space_info::available is operating system dependent. [Note:
available may be less than free. — end note]

29.11.14.35 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: 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: 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: 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: 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: 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 (29.11.9.2), 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.

29.11.14.36 Status known

bool status_known(file_status s) noexcept;
1 Returns: s.type() != file_type::none.

29.11.14.37 Symlink status

file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;
1 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.
2 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.

Remarks: Pathname resolution terminates if p names a symbolic link.

Throws: As specified in 29.11.6.

29.11.14.38 Temporary directory path

path temp_directory_path();
path temp_directory_path(error_code& ec);

Let p be an unspecified directory path suitable for temporary files.

Effects: If exists(p) is false or is_directory(p) is false, an error is reported (29.11.6).

Returns: The path p. The signature with argument ec returns path() if an error occurs.

Throws: As specified in 29.11.6.

[Example: 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]

29.11.14.39 Weakly canonical

path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);

Returns: p with symlinks resolved and the result normalized (29.11.7.1).

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 (29.11.7.1).

Remarks: Implementations should avoid unnecessary normalization such as when canonical has already
been called on the entirety of p.

Throws: As specified in 29.11.6.

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
The contents and meaning of the header `<cstdio>` are the same as the C standard library header `<stdio.h>`.

Calls to the function `tmpnam` with an argument that is a null pointer value may introduce a data race (16.5.5.10) with other calls to `tmpnam` with an argument that is a null pointer value.

**See also:** ISO C 7.21

### 29.12.2 Header `<cinttypes>` synopsis

```
#include <cinttypes>  // see 17.4.1
```

```
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 wcstol(const wchar_t* nptr, wchar_t** endptr, int base);
uintmax_t wcstoul(const wchar_t* nptr, wchar_t** endptr, int base);

intmax_t abs(intmax_t);
// optional, see below
imaxdiv_t div(intmax_t, intmax_t);  // optional, see below

#define PRIxN see below
#define PRIuN see below
#define PRIoN see below
#define PRIuN see below
#define PRIdN see below
#define PRIiN see below
#define PRIoN see below
#define PRIuN 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 SCNxLEASTN see below
#define PRIiFASTN see below
#define PRIoFASTN see below
#define PRIuFASTN see below
#define PRIxFASTN see below
#define PRIxFASTN see below
#define SCNdFASTN see below
#define SCNiFASTN see below
#define SCNoFASTN see below
#define SCNuFASTN see below
#define SCNxFASTN see below
#define PRIiMAX see below
#define PRIoMAX see below
#define PRIuMAX see below
#define PRIxMAX see below
#define SCNdMAX see below
#define SCNiMAX see below
#define SCNoMAX see below
#define SCNuMAX see below
#define SCNxMAX see below
#define PRIdPTR see below
#define PRIiPTR see below
#define PRIoPTR see below
#define PRIuPTR see below
#define PRIxPTR see below
#define PRIXPTR see below
#define SCNdPTR see below
#define SCNiPTR see below
#define SCNoPTR see below
#define SCNuPTR see below
#define SCNxPTR see below

§ 29.12.2
The contents and meaning of the header `<cinttypes>` are the same as the C standard library header `<inttypes.h>`, with the following changes:

1. The header `<cinttypes>` includes the header `<cstdint>` (17.4.1) instead of `<stdint.h>`, and
2. if and only if the type `intmax_t` designates an extended integer type (6.8.1), the following function signatures are added:

   ```
   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>Definitions</td>
</tr>
<tr>
<td>30.3</td>
<td>Requirements</td>
</tr>
<tr>
<td>30.5</td>
<td>Constants</td>
</tr>
<tr>
<td></td>
<td>&lt;regex&gt;</td>
</tr>
<tr>
<td>30.6</td>
<td>Exception type</td>
</tr>
<tr>
<td>30.7</td>
<td>Traits</td>
</tr>
<tr>
<td>30.8</td>
<td>Regular expression template</td>
</tr>
<tr>
<td>30.9</td>
<td>Submatches</td>
</tr>
<tr>
<td>30.10</td>
<td>Match results</td>
</tr>
<tr>
<td>30.11</td>
<td>Algorithms</td>
</tr>
<tr>
<td>30.12</td>
<td>Iterators</td>
</tr>
<tr>
<td>30.13</td>
<td>Grammar</td>
</tr>
</tbody>
</table>

30.2 Definitions [re.def]

1 The following definitions shall apply to this Clause:

30.2.1 collating element [defns.regex.collating.element]
a sequence of one or more characters within the current locale that collate as if they were a single character.

30.2.2 finite state machine [defns.regex.finite.state.machine]
an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

30.2.3 format specifier [defns.regex.format.specifier]
a sequence of one or more characters that is to be replaced with some part of a regular expression match.

30.2.4 matched [defns.regex.matched]
a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

30.2.5 primary equivalence class [defns.regex.primary.equivalence.class]
a 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.
30.2.6 [defns.regex.regular.expression]
regular expression
a pattern that selects specific strings from a set of character strings.

30.2.7 [defns.regex.subexpression]
sub-expression
a subset of a regular expression that has been marked by parenthesis.

30.3 Requirements [re.req]

1 This subclause defines requirements on classes representing regular expression traits. [Note: The class template regex_traits, defined in 30.7, meets these requirements. —end note]

2 The class template basic_regex, defined in 30.8, 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 (23.3.5.2); F1 and F2 are forward iterators (23.3.5.4); 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>
<tr>
<td>X::char_class_type</td>
<td>A bitmask type (16.4.2.2.4).</td>
<td>A bitmask type representing a particular character classification.</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>
</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>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: The value of I will only be 8, 10, or 16. —end note]</td>
</tr>
<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>

[Note: 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 30.7. — end note]

30.4 Header <regex> synopsis

```cpp
#include <compare>    // see 17.11.1
#include <initializer_list>    // see 17.10.1

namespace std {
    // 30.5, regex constants
    namespace regex_constants {
        using syntax_option_type = T1;
        using match_flag_type = T2;
        using error_type = T3;
    }

    // 30.6, class regex_error
    class regex_error;
}
```

§ 30.4 1488
// 30.7, class template regex_traits
template<class charT> struct regex_traits;

// 30.8, class template basic_regex
template<class charT, class traits = regex_traits<charT>> class basic_regex;

using regex = basic_regex<char>;
using uregex = basic_regex<wchar_t>;

// 30.8.5, basic_regex swap
template<class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

// 30.9, 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 assub_match = sub_match<string::const_iterator>
using wssub_match = sub_match<wstring::const_iterator>

// 30.9.2, 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);

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);

// 30.10, 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>;
template<class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
               const match_results<BidirectionalIterator, Allocator>& m2);

template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

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 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 charT, class Allocator, class traits>
bool regex_match(const charT* str,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default) = delete;

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);
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);

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);

// 30.11.4, 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);

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);

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>
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);

// 30.12.1, 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>;

// 30.12.2, 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>;
}

30.5 Namespace std::regex_constants

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.

30.5.1 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;
}

The type syntax_option_type is an implementation-defined bitmask type (16.4.2.2.4). 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.

30.5.2 Bitmask type match_flag_type

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;
}

§ 30.5.2 1492
Table 136: syntax_option_type effects  [tab:re.synopt]

<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 30.13. 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>

```
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 (16.4.2.2.4). 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).  [tab:re.matchflag]

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_not_bol</td>
<td>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).</td>
</tr>
</tbody>
</table>
Table 137: regex_constants::match_flag_type effects when obtaining a match against a character container sequence [first, last]. (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_not_eol</td>
<td>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 &quot;$&quot; in the regular expression shall not match (last, last).</td>
</tr>
<tr>
<td>match_not_bow</td>
<td>The expression &quot;\b&quot; shall not match the sub-sequence (first, first).</td>
</tr>
<tr>
<td>match_not_eow</td>
<td>The expression &quot;\b&quot; shall not match the sub-sequence (last, last).</td>
</tr>
<tr>
<td>match_any</td>
<td>If more than one match is possible then any match is an acceptable result.</td>
</tr>
<tr>
<td>match_not_null</td>
<td>The expression shall not match an empty sequence.</td>
</tr>
<tr>
<td>match_continuous</td>
<td>The expression shall only match a sub-sequence that begins at first.</td>
</tr>
<tr>
<td>match_prev_avail</td>
<td>--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 (30.11) and iterators (30.12).</td>
</tr>
<tr>
<td>format_default</td>
<td>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.</td>
</tr>
<tr>
<td>format_sed</td>
<td>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.</td>
</tr>
<tr>
<td>format_no_copy</td>
<td>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.</td>
</tr>
<tr>
<td>format_first_only</td>
<td>When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.</td>
</tr>
</tbody>
</table>

### 30.5.3 Implementation-defined error_type

```cpp
namespace std::regex_constants {
    using error_type = T3;
    inline constexpr error_type error_collate = unspecified;
    inline constexpr error_type error_ctype = unspecified;
    inline constexpr error_type error_escape = unspecified;
    inline constexpr error_type error_backref = unspecified;
    inline constexpr error_type error_brack = unspecified;
    inline constexpr error_type error_paren = unspecified;
    inline constexpr error_type error_brace = unspecified;
    inline constexpr error_type error_badbrace = unspecified;
    inline constexpr error_type error_range = unspecified;
    inline constexpr error_type error_complexity = unspecified;
    inline constexpr error_type error_stack = unspecified;
}
```

1. The type `error_type` is an implementation-defined enumerated type (16.4.2.2.3). 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>error_collate</td>
<td>The expression contained an invalid collating element name.</td>
</tr>
<tr>
<td>error_ctype</td>
<td>The expression contained an invalid character class name.</td>
</tr>
<tr>
<td>error_escape</td>
<td>The expression contained an invalid escaped character, or a trailing escape.</td>
</tr>
<tr>
<td>error_backref</td>
<td>The expression contained an invalid back reference.</td>
</tr>
<tr>
<td>error_brack</td>
<td>The expression contained mismatched [ and ].</td>
</tr>
<tr>
<td>error_paren</td>
<td>The expression contained mismatched ( and ).</td>
</tr>
</tbody>
</table>
Table 138: error_type values in the C locale (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_brace</td>
<td>The expression contained mismatched { and }</td>
</tr>
<tr>
<td>error_bradbrace</td>
<td>The expression contained an invalid range in a {} expression.</td>
</tr>
<tr>
<td>error_range</td>
<td>The expression contained an invalid character range, such as [b-a] in most encodings.</td>
</tr>
<tr>
<td>error_space</td>
<td>There was insufficient memory to convert the expression into a finite state machine.</td>
</tr>
<tr>
<td>error_badbrace</td>
<td>One of *?+{ was not preceded by a valid regular expression.</td>
</tr>
<tr>
<td>error_complexity</td>
<td>The complexity of an attempted match against a regular expression exceeded a pre-set level.</td>
</tr>
<tr>
<td>error_stack</td>
<td>There was insufficient memory to determine whether the regular expression could match the specified character sequence.</td>
</tr>
</tbody>
</table>

30.6 Class regex_error

class regex_error : public runtime_error {
    public:
        explicit regex_error(regex_constants::error_type ecode);
        regex_constants::error_type code() const;
    };

The class regex_error defines the type of objects thrown as exceptions to report errors from the regular expression library.

regex_error(regex_constants::error_type ecode);

Postconditions: ecode == code().

regex_constants::error_type code() const;

Returns: The error code that was passed to the constructor.

30.7 Class template regex_traits

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;
    };
}
The specializations `regex_traits<char>` and `regex_traits<wchar_t>` meet the requirements for a regular expression traits class (30.3).

```cpp
using char_class_type = bitmask_type;
```

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`.

```cpp
static size_t length(const char_type* p);
```

Returns: `char_traits<charT>::length(p)`.

```cpp
charT translate(charT c) const;
```

Returns: `c`.

```cpp
charT translate_nocase(charT c) const;
```

Returns: `use_facet<ctype<charT>>(getloc()).tolower(c)`.

```cpp
template<class ForwardIterator>
string_type transform(ForwardIterator first, ForwardIterator last) const;
```

Effects: As if by:

```cpp
string_type str(first, last);
return use_facet<collate<charT>>(getloc()).transform(str.data(), str.data() + str.length());
```

```cpp
template<class ForwardIterator>
string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
```

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.

```cpp
template<class ForwardIterator>
string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
```

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.

```cpp
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.\(^{325}\) 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.

```cpp
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);
```

\(^{325}\) For example, if the parameter `icase` is `true` then `[:lower:]` is the same as `[:alpha:]`. 

§ 30.7
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 cctype<charT>& ct = use_facet<cctype<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:**

```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 == cctype_base::digit|cctype_base::upper
```

**Example:**

```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('_', f); // returns true
t.isctype(' ', f); // returns false
```

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.

locale_type imbue(locale_type loc);

**Effects:** Imbues this with a copy of the locale `loc`. [Note: Calling `imbue` with a different locale than the one currently in use invalidates all cached data held by *this. — end note]

**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`.

**Postconditions:** `getloc() == loc`.

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`.

### 30.8 Class template `basic_regex`[re.regex]

1. 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 30.8, `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`.

2. 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: Implementations will typically declare some function templates as friends of `basic_regex` to achieve this. — end note]

3. The functions described in this Clause report errors by throwing exceptions of type `regex_error`. 
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>

namespace std {
  template<class charT, class traits = regex_traits<charT>>
  class basic_regex {
    public:
      // types
      using value_type = charT;
      using traits_type = traits;
      using string_type = typename traits::string_type;
      using flag_type = regex_constants::syntax_option_type;
      using locale_type = typename traits::locale_type;

      // 30.5.1, 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;

      // 30.8.1, 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&);
      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);
  } // end class basic_regex

§ 30.8
// 30.8.2, 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);
basic_regex& assign(initializer_list<charT>, flag_type f = regex_constants::ECMAScript);

// 30.8.3, const operations
unsigned mark_count() const;
flag_type flags() const;

// 30.8.4, locale
locale_type imbue(locale_type loc);
locale_type getloc() const;

// 30.8.5, 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>;

30.8.1 Constructors [re.regex.construct]

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).

30.8.2 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);

30.8.3 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.

30.8.4 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.

30.8.5 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.

30.8.6 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).

30.9 Class template sub_match

Class template sub_match denotes the sequence of characters matched by a particular marked sub-expression.
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;

};

30.9.1 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).

30.9.2 Non-member operators

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);
4
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);
5
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);
6
Returns: lhs.compare(rhs) == 0.

template<
class BiIter>
auto operator<=>((const sub_match<BiIter>& lhs, 
const typename iterator_traits<BiIter>::value_type* rhs);
7
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);
8
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().

30.10 Class template match_results

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.

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 comparison functions are different from those required for a container.

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.
The \texttt{sub_match} object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the \texttt{sub_match} member \texttt{matched} is always \texttt{true}. The \texttt{sub_match} object stored at index \texttt{n} denotes what matched the marked sub-expression \texttt{n} within the matched expression. If the sub-expression \texttt{n} participated in a regular expression match then the \texttt{sub_match} member \texttt{matched} evaluates to \texttt{true}, and members \texttt{first} and \texttt{second} denote the range of characters \{\texttt{first}, \texttt{second}\} which formed that match. Otherwise \texttt{matched} is \texttt{false}, and members \texttt{first} and \texttt{second} point to the end of the sequence that was searched. \[\textit{Note: The \texttt{sub_match} objects representing different sub-expressions that did not participate in a regular expression match need not be distinct. —end note}\]

```cpp
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>;

        // 30.10.1, 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;
        match_results& operator=(const match_results& m);
        match_results& operator=(match_results&& m);
        ~match_results();

        // 30.10.2, state
        bool ready() const;

        // 30.10.3, size
        size_type size() const;
        size_type max_size() const;
        [[nondiscard]] bool empty() const;

        // 30.10.4, element access
        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;

        const_reference prefix() const;
        const_reference suffix() const;
        const_iterator begin() const;
        const_iterator end() const;
        const_iterator cbegin() const;
        const_iterator cend() const;

        // 30.10.5, format
        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;
```
30.10.1 Constructors

```cpp
explicit match_results(const Allocator& a);
```

1. **Postconditions:** ready() returns false, size() returns 0.

```cpp
match_results(match_results&& m) noexcept;
```

2. **Effects:** The stored Allocator value is move constructed from m.get_allocator().

```cpp
Postconditions: As specified in Table 140.
```

```cpp
match_results& operator=(const match_results& m);
```

4. **Postconditions:** As specified in Table 140.

```cpp
match_results& operator=(match_results&& m);
```

5. **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>

30.10.2 State

```cpp
bool ready() const;
```

1. **Returns:** true if *this has a fully established result state, otherwise false.

30.10.3 Size

```cpp
size_type size() const;
```

1. **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:** The state of a match_results object can be modified only by passing that object to regex_match or regex_search.
Subclauses 30.11.2 and 30.11.3 specify the effects of those algorithms on their `match_results` arguments.

--- end note

```cpp
size_type max_size() const;
```

2 *Returns:* The maximum number of `sub_match` elements that can be stored in `*this`.

```cpp
[nodiscard] bool empty() const;
```

3 *Returns:* `size() == 0`.

### 30.10.4 Element access

```cpp
difference_type length(size_type sub = 0) const;
```

1 *Preconditions:* `ready() == true`.

2 *Returns:* `(*this)[sub].length()`.

```cpp
difference_type position(size_type sub = 0) const;
```

3 *Preconditions:* `ready() == true`.

4 *Returns:* The distance from the start of the target sequence to `(*this)[sub].first`.

```cpp
string_type str(size_type sub = 0) const;
```

5 *Preconditions:* `ready() == true`.

6 *Returns:* `string_type((*this)[sub])`.

```cpp
const_reference operator[](size_type n) const;
```

7 *Preconditions:* `ready() == true`.

8 *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.

```cpp
const_reference prefix() const;
```

9 *Preconditions:* `ready() == true`.

10 *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.

```cpp
const_reference suffix() const;
```

11 *Preconditions:* `ready() == true`.

12 *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.

```cpp
const_iterator begin() const;
const_iterator cbegin() const;
```

13 *Returns:* A starting iterator that enumerates over all the sub-expressions stored in `*this`.

```cpp
const_iterator end() const;
const_iterator cend() const;
```

14 *Returns:* A terminating iterator that enumerates over all the sub-expressions stored in `*this`.

### 30.10.5 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;
```

1 *Preconditions:* `ready() == true` and `OutputIter` meets the requirements for a `Cpp17OutputIterator` (23.3.5.3).
Effects: Copies the character sequence \([\text{fmt}\_\text{first}, \text{fmt}\_\text{last})\] to \text{OutputIter} \text{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 \(*\text{this}*\) to which it refers. The bitmasks specified in \text{flags} determine which format specifiers and escape sequences are recognized.

Returns: \text{out}.

\begin{verbatim}
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;
\end{verbatim}

Effects: Equivalent to:

\begin{verbatim}
return format(out, fmt.data(), fmt.data() + fmt.size(), flags);
\end{verbatim}

\begin{verbatim}
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;
\end{verbatim}

Preconditions: \text{ready()} == true.

Effects: Constructs an empty string \text{result} of type \text{basic_string<char_type, ST, SA>} and calls:

\begin{verbatim}
format(back_inserter(result), fmt, flags);
\end{verbatim}

Returns: \text{result}.

\begin{verbatim}
string_type format(
    const char_type* fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;
\end{verbatim}

Preconditions: \text{ready()} == true.

Effects: Constructs an empty string \text{result} of type \text{string_type} and calls:

\begin{verbatim}
format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags);
\end{verbatim}

Returns: \text{result}.

### 30.10.6 Allocator [re.results.all]

\text{allocator_type} \text{get_allocator()} \text{const};

Returns: A copy of the \text{Allocator} that was passed to the object’s constructor or, if that allocator has been replaced, a copy of the most recent replacement.

### 30.10.7 Swap [re.results.swap]

\text{void} \text{swap}(\text{match_results} \& \text{that});

Effects: Swaps the contents of the two sequences.

Postconditions: \text{*this} contains the sequence of matched sub-expressions that were in \text{that}, \text{that} contains the sequence of matched sub-expressions that were in \text{*this}.

Complexity: Constant time.

\begin{verbatim}
template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>\& m1, match_results<BidirectionalIterator, Allocator>\& m2);
\end{verbatim}

Effects: As if by \text{m1.swap(m2)}.

### 30.10.8 Non-member functions [re.results.nonmember]

\begin{verbatim}
template<class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>\& m1, const match_results<BidirectionalIterator, Allocator>\& m2);
\end{verbatim}

Returns: \text{true} if neither match result is ready, \text{false} if one match result is ready and the other is not. If both match results are ready, returns \text{true} only if:
m1.empty() && m2.empty(), or

!m1.empty() && !m2.empty(), and the following conditions are satisfied:

- m1.prefix() == m2.prefix(),
- m1.size() == m2.size() && equal(m1.begin(), m1.end(), m2.begin()), and
- m1.suffix() == m2.suffix().

[Note: The algorithm equal is defined in Clause 25. — end note]

### 30.11 Regular expression algorithms

#### 30.11.1 Exceptions

The algorithms described in subclause 30.11 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*.

#### 30.11.2 regex_match

```c++
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);
```

**Preconditions:** *BidirectionalIterator* meets the `Cpp17BidirectionalIterator` requirements (23.3.5.5).

**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]**

```c++
std::regex re("Get\|GetValue");
std::cmatch m;
regex_search("GetValue", m, re); // returns true, and m[0] contains "Get"
regex_match("GetValue", m, re); // returns true, and m[0] contains "GetValue"
regex_search("GetValues", m, re); // returns true, and m[0] contains "Get"
regex_match("GetValues", m, re); // returns false
```

**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>
<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>first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>false</td>
</tr>
<tr>
<td>m[0].first</td>
<td>first</td>
</tr>
<tr>
<td>m[0].second</td>
<td>last</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>
</tbody>
</table>
Table 141: Effects of regex_match algorithm (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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 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);

Effects: Behaves “as if” by constructing an instance of match_results<BidirectionalIterator>
what, and then returning the result of regex_match(first, last, what, e, flags).

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);

Returns: regex_match(str, str + char_traits<charT>::length(str), m, e, flags).

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);

Returns: regex_match(s.begin(), s.end(), m, e, flags).

30.11.3 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 Cpp17BidirectionalIterator requirements (23.3.5.5).

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><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>m[0].first</code></td>
</tr>
<tr>
<td><code>m.prefix().matched</code></td>
<td><code>m.prefix().first != m.prefix().second</code></td>
</tr>
<tr>
<td><code>m.suffix().first</code></td>
<td><code>m[0].second</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>m.suffix().first != m.suffix().second</code></td>
</tr>
<tr>
<td><code>m[0].first</code></td>
<td>The start of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td><code>m[0].second</code></td>
<td>The end of the sequence of characters that matched the regular expression</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 <code>last</code>.</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 <code>last</code>.</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>

```
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);

4 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);

5 Returns: `regex_search(s.begin(), s.end(), m, e, flags)`. 

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 Allocator, class traits>
bool regex_search(const charT* str, 
                  const basic_regex<charT, traits>& e, 
                  regex_constants::match_flag_type flags = regex_constants::match_default);

7 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);

8 Returns: `regex_search(s.begin(), s.end(), e, flags)`. 
```
30.11.4 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);
```

### 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. **If `!(flags & regex_constants::format_no_copy)`**, calls
   
   ```
   out = copy(m.prefix().first, m.prefix().second, out)
   ```

2. **Then calls**
   
   ```
   out = m.format(out, fmt, flags)
   ```
   
   for the first form of the function and
   
   ```
   out = 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

```
out = 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:

```
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,  
```

§ 30.11.4
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);

Effects: Constructs an empty string result of type basic_string<charT> and calls:

    regex_replace(back_inserter(result), s, s + char_traits<charT>::length(s), e, fmt, flags);

Returns: result.

30.12 Regular expression iterators

30.12.1 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_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.

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);

        private:
            BidirectionalIterator begin; // exposition only
            BidirectionalIterator end; // exposition only
            const regex_type* pregex; // exposition only
    }  // namespace std
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: 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]

### 30.12.1.1 Constructors

`regex_iterator();`

Effects: Constructs an end-of-sequence iterator.

```cpp
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.

### 30.12.1.2 Comparisons

```cpp
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. `begin == right.begin`,
2. `end == right.end`,
3. `pregex == right.pregex`,
4. `flags == right.flags`, and
5. `match[0] == right.match[0]`;

otherwise `false`.

### 30.12.1.3 Indirection

```cpp
const value_type& operator*() const;
```

Returns: `match`.

```cpp
const value_type* operator->() const;
```

Returns: `addressof(match)`.

### 30.12.1.4 Increment

```cpp
regex_iterator& operator++();
```

Effects: Constructs a local variable `start` of type `BidirectionalIterator` and initializes it with the value of `match[0].second`.

1. If the iterator holds a zero-length match and `start == end` the operator sets `*this` to the end-of-sequence iterator and returns `*this`.
2. Otherwise, if the iterator holds a zero-length match, the operator calls:

   ```cpp
   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.

3. 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 \texttt{false} the iterator sets \texttt{*this} to the end-of-sequence iterator. The iterator then returns \texttt{*this}.

In all cases in which the call to \texttt{regex_search} returns \texttt{true}, \texttt{match.prefix().first} shall be equal to the previous value of \texttt{match[0].second}, and for each index \texttt{i} in the half-open range \([0, \texttt{match.size()})\) for which \texttt{match[i].matched} is \texttt{true}, \texttt{match.position(i)} shall return \texttt{distance(begin, match[i].first)}.

[\textit{Note:} This means that \texttt{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 \texttt{regex_search}. — end note]

It is unspecified how the implementation makes these adjustments.

[\textit{Note:} This means that a compiler may call an implementation-specific search function, in which case a program-defined specialization of \texttt{regex_search} will not be called. — end note]

\begin{verbatim}
regex_iterator operator++(int);
\end{verbatim}

\textbf{Effects:} As if by:

\begin{verbatim}
regex_iterator tmp = *this;
++(*this);
return tmp;
\end{verbatim}

\subsection{30.12.2 Class template \texttt{regex_token_iterator}[re.tokiter]}

The class template \texttt{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 \texttt{sub_match} class template instance that represents what matched a particular sub-expression within the regular expression.

When class \texttt{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.

After it is constructed, the iterator finds and stores a value \texttt{regex_iterator<BidirectionalIterator> position} and sets the internal count \texttt{N} to zero. It also maintains a sequence \texttt{subs} which contains a list of the sub-expressions which will be enumerated. Every time \texttt{operator++} is used the count \texttt{N} is incremented; if \texttt{N} exceeds or equals \texttt{subs.size()}, then the iterator increments member \texttt{position} and sets count \texttt{N} to zero.

If the end of sequence is reached (\texttt{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.

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 \texttt{operator*} on an end-of-sequence iterator is not defined. For any other iterator value a \texttt{const sub_match<BidirectionalIterator>\&} is returned. The result of \texttt{operator->} on an end-of-sequence iterator is not defined. For any other iterator value a \texttt{const sub_match<BidirectionalIterator>\*} is returned.

It is impossible to store things into \texttt{regex_token_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.

\begin{verbatim}
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*;
  }
\end{verbatim}
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,
const vector<int>& submatches,
regex_constants::match_flag_type m =
regex_constants::match_default) = delete;
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) = delete;
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) = delete;
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) = delete;
regex_token_iterator(const regex_token_iterator&);
regex_token_iterator& operator=(const regex_token_iterator&);
bool operator==(const regex_token_iterator& const);
const value_type* operator*() const;
const value_type* operator->() const;
regex_token_iterator& operator++();
regex_token_iterator operator++(int);

private:
using position_iterator =
regex_iterator<BidirectionalIterator, charT, traits>;
// exposition only
position_iterator position;
// exposition only
const value_type* result;
// exposition only
value_type suffix;
// exposition only
size_t N;
// exposition only
vector<int> subs;
// exposition only
};
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: 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].

### 30.12.2.1 Constructors

regex_token_iterator();

Effects: Constructs the end-of-sequence 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);

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.

### 30.12.2.2 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.

### 30.12.2.3 Indirection

const value_type& operator*() const;

Returns: *result.

const value_type* operator->() const;

Returns: result.
30.12.2.4 Increment

regex_token_iterator& operator++();

1 Effects: Constructs a local variable prev of type position_iterator, initialized with the value of position.
2 If *this is a suffix iterator, sets *this to an end-of-sequence iterator.
3 Otherwise, if \( N + 1 < \text{subs.size()} \), increments \( N \) and sets result to the address of the current match.
4 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.
5 Otherwise, if any of the values stored in subs is equal to -1 and \( \text{prev->suffix().length()} \) is not 0 the operator sets *this to a suffix iterator that points to the range \( [\text{prev->suffix().first}, \text{prev->suffix().second}) \).
6 Otherwise, sets *this to an end-of-sequence iterator.
7 Returns: *this

regex_token_iterator& operator++(int);

8 Effects: Constructs a copy tmp of *this, then calls ++(*this).
9 Returns: tmp.

30.13 Modified ECMAScript regular expression grammar

The regular expression grammar recognized by basic_regex objects constructed with the ECMAScript flag is that specified by ECMA-262, except as specified below.

2 Objects of type specialization of basic_regex store within themselves a default-constructed instance of their traits template parameter, henceforth referred to as traits_inst. This traits_inst object is used to support localization of the regular expression; basic_regex member functions shall not call any 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

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.
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 \[^[_[:alnum:]]\]

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.

The results from multiple calls to traits_inst.lookup_classname can be bitwise OR'ed together and subsequently passed to traits_inst.isctype.

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.

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: This means that values of the form "uxxxxx" 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 (30.5.2) specified when using the regular expression object in one of the regular expression algorithms (30.11). 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()) 
&amp 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

This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

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</td>
<td>Type aliases &lt;atomic&gt;</td>
</tr>
<tr>
<td>31.4</td>
<td>Order and consistency</td>
</tr>
<tr>
<td>31.5</td>
<td>Lock-free property</td>
</tr>
<tr>
<td>31.6</td>
<td>Waiting and notifying</td>
</tr>
<tr>
<td>31.7</td>
<td>Class template atomic_ref</td>
</tr>
<tr>
<td>31.8</td>
<td>Class template atomic</td>
</tr>
<tr>
<td>31.9</td>
<td>Non-member functions</td>
</tr>
<tr>
<td>31.10</td>
<td>Flag type and operations</td>
</tr>
<tr>
<td>31.11</td>
<td>Fences</td>
</tr>
</tbody>
</table>

31.2 Header <atomic> synopsis

```cpp
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;
    // 31.7.4, partial specialization for pointers
    template<class T> struct atomic_ref<T*>;

    // 31.8, class template atomic
    template<class T> struct atomic;
    // 31.8.4, 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>*,
                                   typename atomic<T>::value_type*,
                                   typename atomic<T>::value_type) noexcept;
template<class T>
bool atomic_compare_exchange_weak(atomic<T>*,
                                   typename atomic<T>::value_type*,
                                   typename atomic<T>::value_type) noexcept;
template<class T>
bool atomic_compare_exchange_strong(volatile atomic<T>*,
                                    typename atomic<T>::value_type*,
                                    typename atomic<T>::value_type) noexcept;
template<class T>
bool atomic_compare_exchange_strong(atomic<T>*,
                                    typename atomic<T>::value_type*,
                                    typename atomic<T>::value_type) noexcept;
template<class T>
bool atomic_compare_exchange_weak_explicit(volatile atomic<T>*,
                                           typename atomic<T>::value_type*,
                                           typename atomic<T>::value_type,
                                           memory_order, memory_order) noexcept;
template<class T>
bool atomic_compare_exchange_weak_explicit(atomic<T>*,
                                           typename atomic<T>::value_type*,
                                           typename atomic<T>::value_type,
                                           memory_order, memory_order) noexcept;
template<class T>
bool atomic_compare_exchange_strong_explicit(volatile atomic<T>*,
                                              typename atomic<T>::value_type*,
                                              typename atomic<T>::value_type,
                                              memory_order, memory_order) noexcept;
template<class T>
bool atomic_compare_exchange_strong_explicit(atomic<T>*,
                                              typename atomic<T>::value_type*,
                                              typename atomic<T>::value_type,
                                              memory_order, memory_order) noexcept;
template<class T>
T atomic_fetch_add(volatile atomic<T>* const, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_add(atomic<T>* const, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_add_explicit(volatile atomic<T>* const, typename atomic<T>::difference_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_add_explicit(atomic<T>* const, typename atomic<T>::difference_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_sub(volatile atomic<T>* const, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_sub(atomic<T>* const, typename atomic<T>::difference_type) noexcept;

template<class T>
T atomic_fetch_sub_explicit(volatile atomic<T>* const, typename atomic<T>::difference_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_sub_explicit(atomic<T>* const, typename atomic<T>::difference_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_and(volatile atomic<T>* const, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_and(atomic<T>* const, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_and_explicit(volatile atomic<T>* const, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_and_explicit(atomic<T>* const, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_xor(volatile atomic<T>* const, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_xor(atomic<T>* const, typename atomic<T>::value_type) noexcept;

template<class T>
T atomic_fetch_xor_explicit(volatile atomic<T>* const, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_xor_explicit(atomic<T>* const, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
void atomic_wait(const volatile atomic<T>* const, typename atomic<T>::value_type);

template<class T>
void atomic_wait(const atomic<T>* const, typename atomic<T>::value_type);

template<class T>
void atomic_wait_explicit(const volatile atomic<T>* const, typename atomic<T>::value_type,
memory_order);

template<class T>
void atomic_wait_explicit(const atomic<T>* const, typename atomic<T>::value_type,
memory_order);

template<class T>
void atomic_notify_one(volatile atomic<T>* const);
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_short = atomic<short>;
using atomic_ushort = atomic<unsigned short>;
using atomic_int = atomic<int>;
using atomic_uint = atomic<unsigned int>;
using atomic_long = atomic<long>;
using atomic_ulong = atomic<unsigned long>;
using atomic_llong = atomic<long long>;
using atomic_ullong = atomic<unsigned long long>;
using atomic_char8_t = atomic<char8_t>;
using atomic_char16_t = atomic<char16_t>;
using atomic_char32_t = atomic<char32_t>;
using atomic_wchar_t = atomic<wchar_t>;
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_signed_lock_free = see below;
using atomic_unsigned_lock_free = see below;

// 31.10, flag type and operations
struct atomic_flag;

§ 31.2
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

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.

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: These aliases are optional in freestanding implementations (16.5.1.3). —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

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;
}

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:

(1.1) — memory_order::relaxed: no operation orders memory.

(1.2) — memory_order::release, memory_order::acq_rel, and memory_order::seq_cst: a store operation performs a release operation on the affected memory location.

(1.3) — memory_order::consume: a load operation performs a consume operation on the affected memory location. [Note: Prefer memory_order::acquire, which provides stronger guarantees than memory_order::consume. Implementations have found it infeasible to provide performance better than that of memory_order::acquire. Specification revisions are under consideration. —end note]
— `memory_order::acquire`, `memory_order::acq_rel`, and `memory_order::seq_cst`: a load operation performs an acquire operation on the affected memory location.

[Note: Atomic operations specifying `memory_order::relaxed` are relaxed with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. — end note]

2 An atomic operation $A$ that performs a release operation on an atomic object $M$ synchronizes with an atomic operation $B$ that performs an acquire operation on $M$ and takes its value from any side effect in the release sequence headed by $A$.

3 An atomic operation $A$ on some atomic object $M$ is coherence-ordered before another atomic operation $B$ on $M$ if

- $A$ 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 $B$.

4 There is a single total order $S$ on all `memory_order::seq_cst` operations, including fences, that satisfies the following constraints. First, if $A$ and $B$ are `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 $S$:

- if $A$ and $B$ are both `memory_order::seq_cst` operations, then $A$ precedes $B$ in $S$; and
- if $A$ is a `memory_order::seq_cst` operation and $B$ happens before a `memory_order::seq_cst` fence $Y$, then $A$ precedes $Y$ in $S$; and
- if a `memory_order::seq_cst` fence $X$ happens before $A$ and $B$ is a `memory_order::seq_cst` operation, then $X$ precedes $B$ in $S$; and
- if a `memory_order::seq_cst` fence $X$ happens before $A$ and $B$ happens before a `memory_order::seq_cst` fence $Y$, then $X$ precedes $Y$ in $S$.

[Note: This definition ensures that $S$ is consistent with the modification order of any atomic object $M$. It also ensures that a `memory_order::seq_cst` load $A$ of $M$ gets its value either from the last modification of $M$ that precedes $A$ in $S$ or from some non-`memory_order::seq_cst` modification of $M$ that does not happen before any modification of $M$ that precedes $A$ in $S$. — end note]

[Note: We do not require that $S$ be consistent with “happens before” (6.9.2.1). This allows more efficient implementation of `memory_order::acquire` and `memory_order::release` on some machine architectures. It can produce surprising results when these are mixed with `memory_order::seq_cst` accesses. — end note]

[Note: `memory_order::seq_cst` ensures sequential consistency only for a program that is free of data races and uses exclusively `memory_order::seq_cst` atomic operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In many cases, `memory_order::seq_cst` atomic operations are reordered with respect to other atomic operations performed by the same thread. — end note]

[Note: For example, with $x$ and $y$ initially zero,

```c
// Thread 1:
r1 = y.load(memory_order::relaxed);
x.store(r1, memory_order::relaxed);

// Thread 2:
r2 = x.load(memory_order::relaxed);
y.store(r2, memory_order::relaxed);
```

should not produce $r1 == r2 == 42$, 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: The recommendation similarly disallows \( r1 == r2 == 42 \) in the following example, with \( x \) and \( y \) again initially zero:

```
// Thread 1:
r1 = x.load(memory_order::relaxed);
if (r1 == 42) y.store(42, memory_order::relaxed);
// Thread 2:
r2 = y.load(memory_order::relaxed);
if (r2 == 42) x.store(42, memory_order::relaxed);
```

— end note]

10 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.

11 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

```template<class T>
T kill_dependency(T y) noexcept;
```

12 Effects: The argument does not carry a dependency to the return value (6.9.2).

13 Returns: \( y \).

### 31.5 Lock-free property

[atomics.lockfree]

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.

1 At least one signed integral specialization of the \texttt{atomic} template, along with the specialization for the corresponding unsigned type (6.8.1), is always lock-free. [Note: This requirement is optional in freestanding implementations (16.5.1.3). — end note]

2 The function \texttt{atomic_is_lock_free} (31.8.1) 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.

3 Atomic operations that are not lock-free are considered to potentially block (6.9.2.2).

4 [Note: Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. 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

[atomics.wait]

1 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: 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]

2 [Note: The following functions are atomic waiting operations:

- \texttt{atomic<T>::wait},
- \texttt{atomic_flag::wait},

[§ 31.6]
(2.3) — atomic_wait and atomic_wait_explicit,
(2.4) — atomic_flag_wait and atomic_flag_wait_explicit, and
(2.5) — atomic_ref<T>::wait.

— end note]

3 [Note: The following functions are atomic notifying operations:
(3.1) — atomic<T>::notify_one and atomic<T>::notify_all,
(3.2) — atomic_flag::notify_one and atomic_flag::notify_all,
(3.3) — atomic_notify_one and atomic_notify_all,
(3.4) — atomic_flag_notify_one and atomic_flag_notify_all,
(3.5) — atomic_ref<T>::notify_one and atomic_ref<T>::notify_all.

— end note]

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:
(4.1) — the atomic waiting operation has blocked after observing the result of \( X \),
(4.2) — \( X \) precedes \( Y \) in the modification order of \( M \), and
(4.3) — \( Y \) happens before the call to the atomic notifying operation.

31.7 Class template atomic_ref

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 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;
      void wait(T, memory_order = memory_order::seq_cst) const noexcept;
      void notify_one() const noexcept;
      void notify_all() const noexcept;
  };
}

1 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 (6.9.2.1).
2 The program is ill-formed if is_trivially_copyable_v<T> is false.

§ 31.7 1527
The lifetime (6.7.3) of an object referenced by *ptr shall exceed the lifetime of all atomic_refs 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: 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]

### 31.7.1 Operations

**static constexpr size_t required_alignment;**

The alignment required for an object to be referenced by an atomic reference, which is at least `alignof(T)`.

*Note: 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*

**static constexpr bool is_always_lock_free;**

The static data member is_always_lock_free is true if the atomic_ref type's operations are always lock-free, and false otherwise.

bool is_lock_free() const noexcept;

Returns: true if operations on all objects of the type atomic_ref<T> are lock-free, false otherwise.

atomic_ref(T& obj);

**Preconditions:** The referenced object is aligned to required_alignment.

**Postconditions:** *this references obj.

**Throws:** Nothing.

atomic_ref(const atomic_ref& ref) noexcept;

**Postconditions:** *this references the object referenced by ref.

```cpp
void store(T desired, memory_order order = memory_order::seq_cst) const noexcept;
```

**Preconditions:** The order argument is neither memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.

**Effects:** Atomically replaces the value referenced by *ptr with the value of desired. Memory is affected according to the value of order.

```cpp
T operator=(T desired) const noexcept;
```

**Effects:** Equivalent to:

```
store(desired);
return desired;
```

```cpp
T load(memory_order order = memory_order::seq_cst) const noexcept;
```

**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 referenced by *ptr.

```cpp
operator T() const noexcept;
```

**Effects:** Equivalent to: return load();

```cpp
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 (6.9.2.1) 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: 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.
31.7.2 Specializations for integral types

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>` (17.4.1). For each such type `integral`, the specialization `atomic_ref<integral>` provides additional atomic operations appropriate to integral types. [Note: 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=(integral) const noexcept;
            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.

\[
\text{integral fetch_key(integral operand, memory_order order = memory_order::seq_cst) const noexcept;}
\]

**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 (6.9.2.1).

**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: There are no undefined results arising from the computation. — end note]

\[
\text{integral operator op } = (\text{integral operand}) \text{ const noexcept;}
\]

**Effects:** Equivalent to: return fetch_key(operand) op operand;

### 31.7.3 Specializations for floating-point types

[atomics.ref.float]

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.

```cpp
namespace std {
    template<> struct atomic_ref<floating-point> {
        private:
            floating-point* ptr; // exposition only
        public:
            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;
            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;
    }
```
The following operations perform arithmetic computations. The key, operator, and computation correspondence are identified in Table 144.

```
floating-point fetch_key(floating-point 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 (6.9.2.1).

**Returns:** Atomically, the value referenced by `*ptr` 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.

```
floating-point operator+=(floating-point operand) const noexcept;
```

**Effects:** Equivalent to:

```
return fetch_key(operand) op operand;
```

### 31.7.4 Partial specialization for pointers

```
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;
            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;

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 145.

T* fetch_key(difference_type operand, memory_order order = memory_order::seq_cst) const noexcept;

**Mandates:** T is a complete object type.

**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 (6.9.2.1).

**Returns:** Atomically, the value referenced by *ptr immediately before the effects.

**Remarks:** The result may be an undefined address, but the operations otherwise have no undefined behavior.

T* operator op=(difference_type operand) const noexcept;

**Effects:** Equivalent to: return fetch_key(operand) op operand;

### 31.7.5 Member operators common to integers and pointers to objects

[atomics.ref.memop]

value_type operator++(int) const noexcept;

**Effects:** Equivalent to: return fetch_add(1);

value_type operator--(int) const noexcept;

**Effects:** Equivalent to: return fetch_sub(1);

value_type operator++() const noexcept;

**Effects:** Equivalent to: return fetch_add(1) + 1;

value_type operator--() const noexcept;

**Effects:** Equivalent to: return fetch_sub(1) - 1;

### 31.8 Class template atomic

[atomics.types.generic]

namespace std {
  template<class T> struct atomic {
    using value_type = T;

    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    // 31.8.1, 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_copy_assignable_v<T>, or
(1.5) — is_move_assignable_v<T>

is false. [Note: Type arguments that are not also statically initializable may be difficult to use. — end note]

2 The specialization atomic<bool> is a standard-layout struct.

3 [Note: The representation of an atomic specialization need not have the same size and alignment requirement as its corresponding argument type. — end note]

31.8.1 Operations on atomic types

constexpr atomic() noexcept(is_nothrow_default_constructible_v<T>);

1 Mandates: is_default_constructible_v<T> is true.

Effects: Initializes the atomic object with the value of T(). Initialization is not an atomic operation (6.9.2).

constexpr atomic(T desired) noexcept;

2 Effects: Initializes the object with the value desired. Initialization is not an atomic operation (6.9.2). [Note: 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: 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: 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;

Preconditions: The order argument is neither memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.

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 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;

Preconditions: The order argument is neither memory_order::release nor memory_order::acq_rel.

Constraints: For the volatile overload of this function, is_always_lock_free is true.

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;

§ 31.8.1
```c
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;
```

**Preconditions:** The failure argument is neither memory_order::release nor memory_order::acq-rel.

**Constraints:** For the volatile overload of this function, is_always_lock_free is true.

**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.

**Returns:** The result of the comparison.

*Note:* For example, the effect of compare_exchange_strong on objects without padding bits (6.8) is

```c
if (memcmp(this, &expected, sizeof(*this)) == 0)
    memcpy(this, &desired, sizeof(*this));
else
    memcpy(expected, this, sizeof(*this));
```

*End note* [Example: 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.

```c
expected = current.load();
do {
    desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));
```

*End example* [Example: 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:

```c
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*]

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.

**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: 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:* 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: 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 = 0xC0DEFEFE;
};
atomic<padded> pad = {};

bool zap() {
    padded expected, desired(0, 0);
    return pad.compare_exchange_strong(expected, desired);
}
```

— end note] [Note: 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;

29 Preconditions: order is neither memory_order::release nor memory_order::acq_rel.
30 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.
31 Remarks: This function is an atomic waiting operation (31.6).

void notify_one() volatile noexcept;
void notify_one() noexcept;

32 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.
33 Remarks: This function is an atomic notifying operation (31.6).

void notify_all() volatile noexcept;
void notify_all() noexcept;

34 Effects: Unblocks the execution of all atomic waiting operations that are eligible to be unblocked (31.6)
by this call.
35 Remarks: This function is an atomic notifying operation (31.6).

31.8.2 Specializations for integers [atomics.types.int]

1 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, unsigned long
long, char8_t, char16_t, char32_t, wchar_t, and any other types needed by the typedefs in the header
<cstdint> (17.4.1). For each such type integral, the specialization atomic<integral> provides additional
atomic operations appropriate to integral types. [Note: The specialization \texttt{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;

    constexpr atomic() noexcept;
    constexpr atomic(integral) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    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 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;

    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_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;

  };

§ 31.8.2
integral operator--() noexcept;
integral operator+=(integral) volatile noexcept;
integral operator+=(integral) noexcept;
integral operator-=integral volatile noexcept;
integral operator-=integral noexcept;
integral operator&=(integral) volatile noexcept;
integral operator&=(integral) noexcept;
integral operator|=(integral) volatile noexcept;
integral operator|=(integral) noexcept;
integral operator^=(integral) volatile noexcept;
integral operator^=(integral) noexcept;

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() noexcept;
void notify_all() volatile noexcept;
void notify_all() noexcept;
};

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>

5 T fetch_key(T operand, memory_order order = memory_order::seq_cst) volatile noexcept;
6 T fetch_key(T operand, memory_order order = memory_order::seq_cst) noexcept;
5 Constraints: For the volatile overload of this function, is_always_lock_free is true.
6 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).
7 Returns: Atomically, the value pointed to by this immediately before the effects.
8 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: There are no undefined results arising from the computation. —end note]
9 T operator op=(T operand) volatile noexcept;
10 T operator op=(T operand) noexcept;
9 Constraints: For the volatile overload of this function, is_always_lock_free is true.
10 Effects: Equivalent to: return fetch_key(operand) op operand;

31.8.3 Specializations for floating-point types

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.

```c++
namespace std {
    template<> struct atomic<floating-point> {
        using value_type = floating-point;
        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(floating-point) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;

void store(floating-point, memory_order = memory_order::seq_cst) volatile noexcept;
void store(floating-point, memory_order = memory_order::seq_cst) noexcept;
floating-point operator=(floating-point) volatile noexcept;
floating-point operator=(floating-point) noexcept;
floating-point load(memory_order = memory_order::seq_cst) volatile noexcept;
floating-point load(memory_order = memory_order::seq_cst) noexcept;
operator floating-point() volatile noexcept;
operator floating-point() noexcept;

floating-point exchange(floating-point,
    memory_order = memory_order::seq_cst) volatile noexcept;
floating-point exchange(floating-point,
    memory_order = memory_order::seq_cst) noexcept;
bool compare_exchange_weak(floating-point&, floating-point,
    memory_order, memory_order) volatile noexcept;
bool compare_exchange_weak(floating-point&, floating-point,
    memory_order, memory_order) noexcept;
bool compare_exchange_strong(floating-point&, floating-point,
    memory_order, memory_order) volatile noexcept;
bool compare_exchange_strong(floating-point&, floating-point,
    memory_order, memory_order) noexcept;

floating-point fetch_add(floating-point,
    memory_order = memory_order::seq_cst) volatile noexcept;
floating-point fetch_add(floating-point,
    memory_order = memory_order::seq_cst) noexcept;
floating-point fetch_sub(floating-point,
    memory_order = memory_order::seq_cst) volatile noexcept;
floating-point fetch_sub(floating-point,
    memory_order = memory_order::seq_cst) noexcept;

floating-point operator+=(floating-point) volatile noexcept;
floating-point operator+=(floating-point) noexcept;
floating-point operator-=(floating-point) volatile noexcept;
floating-point operator-=(floating-point) noexcept;

void wait(floating-point, memory_order = memory_order::seq_cst) const volatile noexcept;
void wait(floating-point, 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;

2 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 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;

**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.

### 31.8.4 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;

        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* 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;

        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, T*, memory_order, memory_order) volatile noexcept;
        bool compare_exchange_weak(T*& T, T*, memory_order, memory_order) noexcept;
        bool compare_exchange_strong(T*& T, T*, memory_order, memory_order) volatile noexcept;
        bool compare_exchange_strong(T*& 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;

T* fetch_add(ptrdiff_t, memory_order = memory_order::seq_cst) volatile noexcept;
T* fetch_add(ptrdiff_t, memory_order = memory_order::seq_cst) noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order::seq_cst) volatile noexcept;
T* fetch_sub(ptrdiff_t, memory_order = memory_order::seq_cst) noexcept;

T* operator++(int) volatile noexcept;
T* operator++(int) noexcept;
T* operator--(int) volatile noexcept;
T* operator--(int) noexcept;
T* operator++() volatile noexcept;
T* operator++() noexcept;
T* operator--() volatile noexcept;
T* operator--() noexcept;
T* operator+=(ptrdiff_t) volatile noexcept;
T* operator+=(ptrdiff_t) noexcept;
T* operator-=(ptrdiff_t) volatile noexcept;
T* operator-=(ptrdiff_t) 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 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.
2 Descriptions are provided below only for members that differ from the primary template.
3 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;

4 Constraints: For the volatile overload of this function, is_always_lock_free is true.
5 Mandates: T is a complete object type. [Note: Pointer arithmetic on void* or function pointers is ill-formed. — end note]
6 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).
7 Returns: Atomically, the value pointed to by this immediately before the effects.
8 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;

31.8.5 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;

31.8.6 Partial specializations for smart pointers
[util.smartptr.atomic]
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 this subclause, 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: 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:

```cpp
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>();
```}

§ 31.8.6
31.8.6.1 Partial specialization for shared_ptr

```cpp
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;
    };
}
```

**Effects:**

1. Initializes `p{}`.

2. Initializes the object with the value `desired`. Initialization is not an atomic operation (6.9.2). *Note: 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*

3. `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`.*

4. 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 (6.9.2.1).

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 31.8.1.

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 fail_order is the same as 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.

bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;

Effects: Equivalent to: return compare_exchange_strong(expected, desired, order, fail_order);

where fail_order is the same as 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.

void wait(shared_ptr<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:

— Evaluates load(order) and compares it to old.

§ 31.8.6.1
If the two are not equivalent, returns.

Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

Remarks: Two `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).

```cpp
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).

```cpp
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).

### 31.8.6.2 Partial specialization for `weak_ptr` [util.smartptr.atomic.weak]

```cpp
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;
        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;
        void wait(weak_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
        void notify_one() noexcept;
        void notify_all() noexcept;
    };

    constexpr atomic() noexcept;
```


2. Effects: Initializes the object with the value `desired`. Initialization is not an atomic operation (6.9.2). [Note: 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

```cpp
void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

3

**Preconditions:** `order` is neither `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

4

**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`.

```cpp
void operator=(weak_ptr<T> desired) noexcept;
```

5

**Effects:** Equivalent to `store(desired)`.

```cpp
weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
```

6

**Preconditions:** `order` is neither `memory_order::release` nor `memory_order::acq_rel`.

7

**Effects:** Memory is affected according to the value of `order`.

**Returns:** Atomically returns `p`.

```cpp
operator weak_ptr<T>() const noexcept;
```

8

**Effects:** Equivalent to: `return load()`;

```cpp
weak_ptr<T> exchange(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

9

**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 (6.9.2.1).

**Returns:** Atomically returns the value of `p` immediately before the effects.

```cpp
bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired, memory_order success, memory_order failure) noexcept;
```

10

**Preconditions:** `failure` is neither `memory_order::release` nor `memory_order::acq_rel`.

11

**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 31.8.1.

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.

```cpp
bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

12

**Effects:** Equivalent to:

```
return compare_exchange_weak(expected, desired, order, fail_order);
```

where `fail_order` is the same as `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`.

```cpp
bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

13

**Effects:** Equivalent to:

```

```
return compare_exchange_strong(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}
void wait(weak_ptr<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:

(20.1) Evaluates load(order) and compares it to \texttt{old}.

(20.2) If the two are not equivalent, returns.

(20.3) Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

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).

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() 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).
\end{verbatim}

31.9 Non-member functions \([\text{atomics.nonmembers}]\)

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.

\begin{verbatim}
\[Note: The non-member functions enable programmers to write code that can be compiled as either C or C++, for example in a shared header file. \quad \text{[end note]}\]
\end{verbatim}

31.10 Flag type and operations \([\text{atomics.flag}]\)

namespace std {

struct atomic_flag {
    constexpr atomic_flag() noexcept;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;

    bool test(memory_order = memory_order::seq_cst) const volatile noexcept;
    bool test(memory_order = memory_order::seq_cst) const noexcept;
    bool test_and_set(memory_order = memory_order::seq_cst) volatile noexcept;
    bool test_and_set(memory_order = memory_order::seq_cst) noexcept;
    void clear(memory_order = memory_order::seq_cst) volatile noexcept;
    void clear(memory_order = memory_order::seq_cst) noexcept;

    void wait(bool, memory_order = memory_order::seq_cst) const volatile noexcept;
    void wait(bool, 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;
};

}
The \texttt{atomic\_flag} type provides the classic test-and-set functionality. It has two states, set and clear.

Operations on an object of type \texttt{atomic\_flag} shall be lock-free.  
\textit{[Note: Hence the operations should also be address-free. — end note]}  

The \texttt{atomic\_flag} type is a standard-layout struct. It has a trivial destructor.

\begin{verbatim}
constexpr atomic_flag::atomic_flag() noexcept;
\end{verbatim}

\textit{Effects:} Initializes \this{} to the clear state.

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 \texttt{atomic\_flag\_test}, let \texttt{order} be \texttt{memory\_order::seq\_cst}.

\textit{Preconditions:} \order{} is neither \texttt{memory\_order::release} nor \texttt{memory\_order::acq\_rel}.

\textit{Effects:} Memory is affected according to the value of \order{}.

\textit{Returns:} Atomically returns the value pointed to by \texttt{object} or \this{}.

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;

\textit{Effects:} Atomically sets the value pointed to by \texttt{object} or by \this{} to \texttt{true}. Memory is affected according to the value of \order{}. These operations are atomic read-modify-write operations (6.9.2).

\textit{Returns:} Atomically, the value of the object immediately before the effects.

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;

\textit{Preconditions:} The \texttt{order} argument is neither \texttt{memory\_order::consume}, \texttt{memory\_order::acquire}, nor \texttt{memory\_order::acq\_rel}.

\textit{Effects:} Atomically sets the value pointed to by \texttt{object} or by \this{} to \texttt{false}. Memory is affected according to the value of \order{}.

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 \texttt{atomic\_flag\_wait}, let \texttt{order} be \texttt{memory\_order::seq\_cst}. Let \texttt{flag} be \texttt{object} for the non-member functions and \this{} for the member functions.

\textit{Preconditions:} \order{} is neither \texttt{memory\_order::release} nor \texttt{memory\_order::acq\_rel}.

\textit{Effects:} Repeatedly performs the following steps, in order:

\begin{itemize}
\item Evaluates \texttt{flag\_\_test(order)} \(!=\) \texttt{old}.
\end{itemize}
If the result of that evaluation is `true`, returns.

Blocks until it is unblocked by an atomic notifying operation or is unblocked spuriously.

**Remarks:** This function is an atomic waiting operation (31.6).

```c
void atomic_flag_notify_one(volatile atomic_flag* object) noexcept;
void atomic_flag::notify_one() volatile 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).

```c
void atomic_flag_notify_all(volatile atomic_flag* object) noexcept;
void atomic_flag::notify_all() volatile 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

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`.

**Effects:** Depending on the value of `order`, this operation:

- has no effects, if `order == memory_order::relaxed`;
- is an acquire fence, if `order == memory_order::acquire` or `order == memory_order::consume`;
- is a release fence, if `order == memory_order::release`;
- is both an acquire fence and a release fence, if `order == memory_order::acq_rel`;
- is a sequentially consistent acquire and release fence, if `order == memory_order::seq_cst`.

**Effects:** Equivalent to `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.

**Note:** `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 `atomic_thread_fence`, but the hardware fence instructions that `atomic_thread_fence` would have inserted are not emitted. — end note]
32 Thread support library [thread]

32.1 General [thread.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 146: Thread support library summary [tab:thread.summary]

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>32.3</td>
<td>Stop tokens</td>
</tr>
<tr>
<td>32.4</td>
<td>Threads</td>
</tr>
<tr>
<td>32.5</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td>32.6</td>
<td>Condition variables</td>
</tr>
<tr>
<td>32.7</td>
<td>Semaphores</td>
</tr>
<tr>
<td>32.8</td>
<td>Coordination types</td>
</tr>
<tr>
<td>32.9</td>
<td>Futures</td>
</tr>
</tbody>
</table>

32.2 Requirements [thread.req]

32.2.1 Template parameter names [thread.req.paramname]

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 [thread.req.exception]

Some functions described in this Clause are specified to throw exceptions of type `system_error` (19.5.7). 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 16.5.5.13.

[Example: 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 [thread.req.native]

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: 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 [thread.req.timing]

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.\(^{326}\) 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:

\[(4.1)\] if \( C_a > C_t \), the waiting function should wake as soon as possible, i.e., \( C_a + D_i + D_m \), since the timeout is already satisfied. This specification may result in the total duration of the wait decreasing when measured against a steady clock.

\[(4.2)\] if \( C_a \leq C_t \), the waiting function should not time out until \( \text{Clock}::\text{now()} \) returns a time \( C_n \geq C_t \), i.e., waking at \( C_t + D_i + D_m \). [Note: 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. [Note: Implementations should decrease the duration of the wait when the clock is adjusted forwards. — end note]

5 [Note: 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 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: 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 Cpp17Lockable types \[\text{thread.req.lockable}\]

32.2.5.1 In general \[\text{thread.req.lockable.general}\]

\(^1\) An execution agent is an entity such as a thread that may perform work in parallel with other execution agents. [Note: 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: 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 unique_lock (32.5.4.3), shared_lock (32.5.4.4), scoped_lock (32.5.4.2), lock_guard (32.5.4.1), lock, try_lock (32.5.5), and condition_variable_any (32.6.4) all operate on user-supplied lockable objects. The Cpp17BasicLockable requirements, the Cpp17Lockable requirements, and the 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: The nature of any lock ownership and any synchronization it entails are not part of these requirements. — end note]

32.2.5.2 Cpp17BasicLockable requirements \[\text{thread.req.lockable.basic}\]

\(^1\) 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).

---

\(^{326}\) All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.
m.lock()

2 Effects: 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.

m.unlock()

3 Preconditions: The current execution agent holds a lock on m.
4 Effects: Releases a lock on m held by the current execution agent.
5 Throws: Nothing.

32.2.5.3 Cpp17Lockable requirements [thread.req.lockable.req]

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).

m.try_lock()

2 Effects: 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.
3 Return type: bool.
4 Returns: true if the lock was acquired, false otherwise.

32.2.5.4 Cpp17TimedLockable requirements [thread.req.lockable.timed]

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, rel_time denotes a value of an instantiation of duration (27.5), and abs_time denotes a value of an instantiation of time_point (27.6)).

m.try_lock_for(rel_time)

2 Effects: Attempts to acquire a lock for the current execution agent within the relative timeout (32.2.4) specified by rel_time. The function will not return within the timeout specified by rel_time unless it has obtained a lock on m for the current execution agent. If an exception is thrown then a lock has not been acquired for the current execution agent.
3 Return type: bool.
4 Returns: true if the lock was acquired, false otherwise.

m.try_lock_until(abs_time)

5 Effects: Attempts to acquire a lock for the current execution agent before the absolute timeout (32.2.4) specified by abs_time. The function will not return before the timeout specified by abs_time unless it has obtained a lock on m for the current execution agent. If an exception is thrown then a lock has not been acquired for the current execution agent.
6 Return type: bool.
7 Returns: true if the lock was acquired, false otherwise.

32.3 Stop tokens [thread.stoptoken]

32.3.1 Introduction [thread.stoptoken.intro]

This clause 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.

stop_source, stop_token, and stop_callback implement semantics of shared ownership of a stop state. Any stop_source, stop_token, or stop_callback that shares ownership of the same stop state is an associated stop_source, stop_token, or stop_callback, respectively. The last remaining owner of the stop state automatically releases the resources associated with the stop state.

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).

4 Callbacks registered via a `stop_callback` object are called when a stop request is first made by any associated `stop_source` object.

5 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`

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:
      // 32.3.3.1, 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;

      // 32.3.3.2, 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;
  }
}
```

#### 32.3.3.1 Constructors, copy, and assignment

```cpp
stop_token() noexcept;
```

Postconditions: `stop_possible()` is `false` and `stop_requested()` is `false`. [Note: 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: *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.

32.3.3.2 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.

32.3.3.3 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

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 {
public:
    // 32.3.4.1, constructors, copy, and assignment
    stop_source();
    explicit stop_source(nostopstate_t) noexcept;

§ 32.3.4 1555

}
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;

// 32.3.4.2, 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;

32.3.4.1 Constructors, copy, and assignment

stop_source();

Effects: Initialises *this to have ownership of a new stop state.

Postconditions: stop_possible() is true and stop_requested() is false.

Throws: bad_alloc if memory could not be allocated for the stop state.

explicit stop_source(nostopstate_t) noexcept;

Postconditions: stop_possible() is false and stop_requested() is false. [Note: No resources are allocated for the state. — end note]

stop_source(const stop_source& rhs) noexcept;

Postconditions: *this == rhs is true. [Note: *this and rhs share the ownership of the same stop state, if any. — end note]

stop_source(stop_source&& rhs) noexcept;

Postconditions: *this contains the value of rhs prior to the start of construction and rhs.stop_possible() is false.

~stop_source();

Effects: Releases ownership of the stop state, if any.

stop_source& operator=(const stop_source& rhs) noexcept;

Effects: Equivalent to: stop_source(rhs).swap(*this).

Returns: *this.

stop_source& operator=(stop_source&& rhs) noexcept;

Effects: Equivalent to: stop_source(std::move(rhs)).swap(*this).

Returns: *this.

void swap(stop_source& rhs) noexcept;

Effects: Exchanges the values of *this and rhs.

32.3.4.2 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.

bool request_stop() noexcept;

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 (6.9.2.1). 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 (14.6.1). [Note: A stop request includes notifying all condition variables of type condition_variable_any temporarily registered during an interruptible wait (32.6.4.2). — end note]

Postconditions: stop_possible() is false or stop_requested() is true.

Returns: true if this call made a stop request; otherwise false.

32.3.4.3 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

namespace std {
    template<class Callback>
    class stop_callback {
    public:
        using callback_type = Callback;

        // 32.3.5.1, 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;

        private:
            Callback callback; // exposition only
    };

    template<class Callback>
    stop_callback(stop_token, Callback) -> stop_callback<Callback>;
}

Mandates: stop_callback is instantiated with an argument for the template parameter Callback that satisfies both invocable and destructible.

Preconditions: stop_callback is instantiated with an argument for the template parameter Callback that models both invocable and destructible.
32.3.5.1 Constructors and destructor

```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>);
```

**Constraints:** Callback and C satisfy `constructible_from<Callback, C>`.

**Preconditions:** Callback and C model `constructible_from<Callback, C>`.

**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`.

**Remarks:** If evaluating `std::forward<Callback>(callback)()` exits via an exception, then `terminate` is called (14.6.1).

**Throws:** Any exception thrown by the initialization of callback.

```
~stop_callback();
```

**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 (6.9.2.1) `callback` is destroyed. If `callback` is executing on the current thread, then the destructor does not block (3.6) waiting for the return from the invocation of `callback`. Releases ownership of the stop state, if any.

32.4 Threads

32.4 describes components that can be used to create and manage threads. [Note: These threads are intended to map one-to-one with operating system threads. —end note]

32.4.1 Header `<thread>` synopsis

```cpp
#include <compare>  // see 17.11.1
#include <initializer_list>  // see 17.10.1
```

```cpp
namespace std {
    class thread {
    public:
        void swap(thread& x, thread& y) noexcept;

        // 32.4.3 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);
        }
    }
}
```

32.4.2 Class thread

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: 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]

namespace std {
    class thread {
        public:
            // types
            class id;
            using native_handle_type = implementation-defined;       // see 32.2.3

            // construct/copy/destroy
            thread() noexcept;
            template<class F, class... Args> explicit thread(F&& f, Args&&... args);
            ~thread();
            thread(const thread&) = delete;
            thread(thread&) noexcept;
            thread& operator=(const thread&) = delete;
            thread& operator=(thread&) noexcept;

            // members
            void swap(thread&) noexcept;
            bool joinable() const noexcept;
            void join();
            void detach();
            id get_id() const noexcept;
            native_handle_type native_handle();       // see 32.2.3

            // static members
            static unsigned int hardware_concurrency() noexcept;
    };
}

32.4.2.1 Class thread::id

namespace std {
    class thread::id {
        public:
            id() noexcept;
    };

    bool operator==(thread::id x, thread::id y) noexcept;
    strong_ordering operator<=>(thread::id x, thread::id y) noexcept;

    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& out, thread::id id);

    // hash support
    template<class T>
    struct hash;
    template<>
    struct hash<thread::id>;
}

1 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 (32.4.2). 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.

2 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.

3 [Note: Relational operators allow thread::id objects to be used as keys in associative containers. — end note]
id() noexcept;

Postconditions: The constructed object does not represent a thread of execution.

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.

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.

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.

template<> struct hash<thread::id>;
The specialization is enabled (20.14.18).

32.4.2.2 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>>,
(4.4) — (is_move_constructible_v<decay_t<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

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: 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:

(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.
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.

32.4.2.3 Destructor
~thread();

Effects: If joinable(), calls terminate(). Otherwise, has no effects. [Note: 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. Thus the programmer must ensure that the destructor is never executed while the thread is still joinable. — end note]

32.4.2.4 Assignment
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.

32.4.2.5 Members
void swap(thread& x) noexcept;

Effects: Swaps the state of *this and x.

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: 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.

### 32.4.2.6 Static members

unsigned hardware_concurrency() noexcept;

Returns: The number of hardware thread contexts. [Note: 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.

### 32.4.2.7 Specialized algorithms

void swap(thread& x, thread& y) noexcept;

Effects: As if by x.swap(y).

### 32.4.3 Class jthread

The class jthread provides a mechanism to create a new thread of execution. The functionality is the same as for class thread (32.4.2) with the additional abilities to provide a stop_token (32.3) to the new thread of execution, make stop requests, and automatically join.

```cpp
namespace std {
    class jthread {
        public:
            // types
            using id = thread::id;
            using native_handle_type = thread::native_handle_type;

            // 32.4.3.1, constructors, move, and assignment
            jthread() noexcept;
            template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
            ~jthread();
            jthread(const jthread&) = delete;
            jthread(jthread&&) noexcept;
            jthread& operator=(const jthread&);
            jthread& operator=(jthread&&) noexcept;

            // 32.4.3.2, members
            void swap(jthread&) noexcept;
            [[nodiscard]] bool joinable() const noexcept;
            void join();
            void detach();
            [[nodiscard]] id get_id() const noexcept;
            [[nodiscard]] native_handle_type native_handle(); // see 32.2.3

            // 32.4.3.3, stop token handling
            [[nodiscard]] stop_source get_stop_source() noexcept;
            [[nodiscard]] stop_token get_stop_token() const noexcept;
            bool request_stop() noexcept;

            // 32.4.3.4, specialized algorithms
            friend void swap(jthread& lhs, jthread& rhs) noexcept;

            // 32.4.3.5, static members
            [[nodiscard]] static unsigned int hardware_concurrency() noexcept;

        private:
            stop_source ssource; // exposition only
        }
    }
}§ 32.4.3

1562
32.4.3.1 Constructors, move, and assignment

jthread() noexcept;

Effects: Constructs a jthread object that does not represent a thread of execution.

Postconditions: get_id() == id() is true and ssoure.stop_possible() is false.

template<class F, class... Args> explicit jthread(F&& f, Args&&... args);

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>...>.

Preconditions: decay_t<F> and each type in decay_t<Args> meet the Cpp17MoveConstructible requirements.

Effects: Initializes ssoure. 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: 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 ssoure.stop_possible() is true and *this represents the newly started thread. [Note: 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: 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. ssoure has the value of x.ssoure prior to the start of construction and x.ssoure.stop_possible() is false.

~jthread();

Effects: If joinable() is true, calls request_stop() and then join(). [Note: 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. ssoure has the value of x.ssoure prior to the assignment and x.ssoure.stop_possible() is false.

Returns: *this.
### 32.4.3.2 Members

```c++
void swap(jthread& x) noexcept;
```

**Effects:** Exchanges the values of *this and x.

```c++
[[nodiscard]] bool joinable() const noexcept;
```

**Returns:** `get_id() != id()`.

```c++
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: Operations on *this are not synchronized. — end note]

**Postconditions:** The thread represented by *this has completed. `get_id() == id()`.

```c++
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:**

- **resource_deadlock_would_occur** — if deadlock is detected or `get_id() == this_thread::get_id()`.
- **no_such_process** — if the thread is not valid.
- **invalid_argument** — if the thread is not joinable.

```c++
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.

### 32.4.3.3 Stop token handling

```c++
[[nodiscard]] stop_source get_stop_source() noexcept;
```

**Effects:** Equivalent to: `return ssource;`

```c++
[[nodiscard]] stop_token get_stop_token() const noexcept;
```

**Effects:** Equivalent to: `return ssource.get_token();`

```c++
bool request_stop() noexcept;
```

**Effects:** Equivalent to: `return ssource.request_stop();`

### 32.4.3.4 Specialized algorithms

```c++
friend void swap(jthread& x, jthread& y) noexcept;
```

**Effects:** Equivalent to: `x.swap(y).

### 32.4.3.5 Static members

```c++
[[nodiscard]] static unsigned int hardware_concurrency() noexcept;
```

**Returns:** `thread::hardware_concurrency()`.

---

§ 32.4.3.5 1564
32.4.4 Namespace this_thread [thread.thread.this]

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 [thread.mutex]

This subclause provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (6.9.2).

32.5.1 Header <mutex> synopsis [mutex.syn]

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;

    § 32.5.1
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);

32.5.2 Header <shared_mutex> synopsis

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;
}

32.5.3 Mutex requirements

32.5.3.1 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.

32.5.3.2 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 this subclause. In this description, m denotes an object of a mutex type.

1. The mutex types meet the Cpp17Lockable requirements (32.2.5.3).

2. 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.

3. 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: This can be viewed as the modification order (6.9.2) of the mutex. —end note] [Note: Construction and destruction of an object of a mutex type need not be thread-safe; other synchronization should be used to ensure that mutex objects are initialized and visible to other threads. —end note]

4. The error conditions for error codes, if any, reported by member functions of the mutex types are as follows:
   (4.1) — resource_unavailable_try_again — if any native handle type manipulated is not available.
   (4.2) — operation_not_permitted — if the thread does not have the privilege to perform the operation.
   (4.3) — invalid_argument — if any native handle type manipulated as part of mutex construction is incorrect.

5. 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:
   (13.1) — operation_not_permitted — if the thread does not have the privilege to perform the operation.
The expression \texttt{m\_try\_lock()} is well-formed and has the following semantics:

\begin{itemize}
  \item \textbf{Preconditions}: If \texttt{m} is of type \texttt{mutex}, \texttt{timed\_mutex}, \texttt{shared\_mutex}, or \texttt{shared\_timed\_mutex}, the calling thread does not own the mutex.
  \item \textbf{Effects}: Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and \texttt{try\_lock()} immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [Note: 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\_lock()} does not consistently return \texttt{false} in the absence of contending mutex acquisitions.
  \item \textbf{Return type}: \texttt{bool}.
  \item \textbf{Returns}: \texttt{true} if ownership of the mutex was obtained for the calling thread, otherwise \texttt{false}.
  \item \textbf{Synchronization}: If \texttt{try\_lock()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (6.9.2) this operation. [Note: Since \texttt{lock()} does not synchronize with a failed subsequent \texttt{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]
  \item \textbf{Throws}: Nothing.
\end{itemize}

The expression \texttt{m\_unlock()} is well-formed and has the following semantics:

\begin{itemize}
  \item \textbf{Preconditions}: The calling thread owns the mutex.
  \item \textbf{Effects}: Releases the calling thread’s ownership of the mutex.
  \item \textbf{Return type}: \texttt{void}.
  \item \textbf{Synchronization}: This operation synchronizes with (6.9.2) subsequent lock operations that obtain ownership on the same object.
  \item \textbf{Throws}: Nothing.
\end{itemize}

32.5.3.2.1 Class \texttt{mutex} \hfill [thread\_mutex\_class]

\begin{verbatim}
namespace std {
    class mutex {
        public:
            constexpr mutex() noexcept;
            ~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
    };
}
\end{verbatim}

1 The class \texttt{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 \texttt{try\_lock()}) or block (for \texttt{lock()}) until the owning thread has released ownership with a call to \texttt{unlock()}.

2 \textbf{Note}: After a thread \texttt{A} has called \texttt{unlock()}, releasing a mutex, it is possible for another thread \texttt{B} to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread \texttt{A} appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread \texttt{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 \texttt{mutex} meets all of the mutex requirements (32.5.3). It is a standard-layout class (11.2).
4 [Note: 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.

32.5.3.2.2 Class recursive_mutex

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 (32.5.3). 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
   (4.2) — a thread terminates while owning a recursive_mutex object.

32.5.3.3 Timed mutex types

1 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).

2 The timed mutex types meet the Cpp17TimedLockable requirements (32.2.5.4).

3 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: 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 \texttt{try\_lock\_for()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (6.9.2) this operation.

\textbf{Throws}: Timeout-related exceptions (32.2.4).

The expression \texttt{m\_try\_lock\_until(abs\_time)} is well-formed and has the following semantics:

\textbf{Preconditions}: If \texttt{m} is of type \texttt{timed\_mutex} or \texttt{shared\_timed\_mutex}, the calling thread does not own the mutex.

\textbf{Effects}: The function attempts to obtain ownership of the mutex. If \texttt{abs\_time} has already passed, the function attempts to obtain ownership without blocking (as if by calling \texttt{try\_lock()}). The function returns before the absolute timeout (32.2.4) specified by \texttt{abs\_time} only if it has obtained ownership of the mutex object. [\textit{Note}: As with \texttt{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]

\textbf{Return type}: \texttt{bool}.

\textbf{Returns}: \texttt{true} if ownership was obtained, otherwise \texttt{false}.

Synchronization: If \texttt{try\_lock\_until()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (6.9.2) this operation.

\textbf{Throws}: Timeout-related exceptions (32.2.4).

\subsection{32.5.3.3.1 Class \texttt{timed\_mutex}}

\begin{verbatim}
namespace std {
    class timed_mutex {
    public:
        timed_mutex();
        ~timed_mutex();
        timed_mutex(const timed_mutex&) = delete;
        timed_mutex& operator=(const timed_mutex&) = delete;
        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();

        using native_handle_type = implementation-defined;  // see 32.2.3
        native_handle_type native_handle();  // see 32.2.3
    }
}
\end{verbatim}

1 The class \texttt{timed\_mutex} provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a \texttt{timed\_mutex} object, attempts by another thread to acquire ownership of that object will fail (for \texttt{try\_lock()} or block (for \texttt{lock()}, \texttt{try\_lock\_for()}, and \texttt{try\_lock\_until()} until the owning thread has released ownership with a call to \texttt{unlock()} or the call to \texttt{try\_lock\_for()} or \texttt{try\_lock\_until()} times out (having failed to obtain ownership).

2 The class \texttt{timed\_mutex} meets all of the timed mutex requirements (32.5.3.3). It is a standard-layout class (11.2).

3 The behavior of a program is undefined if:

\begin{enumerate}
\item it destroys a \texttt{timed\_mutex} object owned by any thread,
\item a thread that owns a \texttt{timed\_mutex} object calls \texttt{lock()}, \texttt{try\_lock()}, \texttt{try\_lock\_for()}, or \texttt{try\_lock\_until()} on that object, or
\item a thread terminates while owning a \texttt{timed\_mutex} object.
\end{enumerate}

\subsection{32.5.3.3.2 Class \texttt{recursive\_timed\_mutex}}

\begin{verbatim}
namespace std {
    class recursive_timed_mutex {
    public:
        recursive_timed_mutex();
        ~recursive_timed_mutex();
    }
}
\end{verbatim}
recursive_timed_mutex();
~recursive_timed_mutex();

recursive_timed_mutex(const recursive_timed_mutex&) = delete;
recursive_timed_mutex& operator=(const recursive_timed_mutex&) = delete;

void lock(); // blocking
bool try_lock() noexcept;
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();

using native_handle_type = implementation-defined; // see 32.2.3
native_handle_type native_handle(); // see 32.2.3
};

1 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).

2 The class recursive_timed_mutex meets all of the timed mutex requirements (32.5.3.3). It is a standard-layout class (11.2).

3 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.

4 The behavior of a program is undefined if:
   (4.1) — it destroys a recursive_timed_mutex object owned by any thread, or
   (4.2) — a thread terminates while owning a recursive_timed_mutex object.

32.5.3.4 Shared mutex types [thread.sharedmutex.requirements]

1 The standard library types shared_mutex and shared_timed_mutex are shared mutex types. Shared mutex types meet the requirements of mutex types (32.5.3.2) and additionally meet the requirements set out below. In this description, m denotes an object of a shared mutex type.

2 In addition to the exclusive lock ownership mode specified in 32.5.3.2, 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.

3 The expression m.lock_shared() is well-formed and has the following semantics:

   Preconditions: The calling thread has no ownership of the mutex.

   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.

   Postconditions: The calling thread has a shared lock on 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.unlock_shared() is well-formed and has the following semantics:

Preconditions: The calling thread holds a shared lock on the mutex.

Effects: Releases a shared lock on the mutex held by the calling thread.

Return type: void.

Synchronization: This operation synchronizes with (6.9.2) subsequent lock() operations that obtain ownership on the same object.

Throws: Nothing.

The expression m.try_lock_shared() is well-formed and has the following semantics:

Preconditions: The calling thread has no ownership of the mutex.

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.

Return type: bool.

Returns: true if the shared ownership lock was acquired, false otherwise.

Synchronization: If try_lock_shared() returns true, prior unlock() operations on the same object synchronize with (6.9.2) this operation.

Throws: Nothing.

32.5.3.4.1 Class shared_mutex

namespace std {
    class shared_mutex {
        public:
            shared_mutex();
            ~shared_mutex();

            shared_mutex(const shared_mutex&) = delete;
            shared_mutex& operator=(const shared_mutex&) = delete;

            // exclusive ownership
            void lock(); // blocking
            bool try_lock();
            void unlock();

            // shared ownership
            void lock_shared(); // blocking
            bool try_lock_shared();
            void unlock_shared();

            using native_handle_type = implementation-defined; // see 32.2.3
            native_handle_type native_handle(); // see 32.2.3
    };
}

The class shared_mutex provides a non-recursive mutex with shared ownership semantics.

The class shared_mutex meets all of the shared mutex requirements (32.5.3.4). It is a standard-layout class (11.2).

The behavior of a program is undefined if:

— it destroys a shared_mutex object owned by any thread,
— a thread attempts to recursively gain any ownership of a shared_mutex, or
— a thread terminates while possessing any ownership of a shared_mutex.

shared_mutex may be a synonym for shared_timed_mutex.

32.5.3.5 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 (32.5.3.3), shared mutex types (32.5.3.4), 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. [Note: 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, false otherwise.

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: 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, false otherwise.

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).

32.5.3.5.1 Class shared_timed_mutex

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 (32.5.3.5). 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.

32.5.4 Locks [thread.lock]

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: Locks are intended to ease the burden of unlocking the lockable object under both normal and exceptional circumstances. —end note]

2 Some lock constructors take tag types which describe what should be done with the lockable object during the lock's construction.

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
    struct adopt_lock_t { };          // without blocking
    // 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 { };         
}

32.5.4.1 Class template lock_guard [thread.lock.guard]

namespace std {
    template<class Mutex>
    class lock_guard {
public:
        using mutex_type = Mutex;

        explicit lock_guard(mutex_type& m);
        lock_guard(mutex_type& m, adopt_lock_t);
        ~lock_guard();

        lock_guard(const lock_guard&) = delete;
        lock_guard& operator=(const lock_guard&) = delete;
    };
}
private:
    mutex_type& pm; // exposition only
};

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`.

Effects: Initializes `pm` with `m`. Calls `m.lock()`.

```cpp
lock_guard(mutex_type& m, adopt_lock_t);
```

4

Preconditions: The calling thread owns the mutex `m`.

Effects: Initializes `pm` with `m`.

Throws: Nothing.

```cpp
~lock_guard();
```

7

Effects: As if by `pm.unlock()`.

### 32.5.4.2 Class template `scoped_lock` [thread.lock.scoped]

```cpp
namespace std {
    template<class... MutexTypes>
    class scoped_lock {
    public:
        using mutex_type = Mutex; // If MutexTypes... consists of the single type Mutex
        explicit scoped_lock(MutexTypes&... m);
        explicit scoped_lock(adopt_lock_t, MutexTypes&... m);
        ~scoped_lock();
        scoped_lock(const scoped_lock&) = delete;
        scoped_lock& operator=(const scoped_lock&) = delete;

    private:
        tuple<MutexTypes&...> pm; // exposition only
    };
}
```

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);
```

2

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);
```

4

Preconditions: The calling thread owns all the mutexes in `m`.

Effects: Initializes `pm` with `tie(m...)`.

Throws: Nothing.
```
~scoped_lock();

7    Effects: For all i in [0, sizeof...(MutexTypes)), get<i>(pm).unlock().

32.5.4.3  Class template unique_lock [thread.lock.unique]

namespace std {
    template<class Mutex>
    class unique_lock {
public:
        using mutex_type = Mutex;

        // 32.5.4.3.1, 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();
        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);

        // 32.5.4.3.2, 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();

        // 32.5.4.3.3, modifiers
        void swap(unique_lock& u) noexcept;
        mutex_type* release() noexcept;

        // 32.5.4.3.4, 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;
    }

1    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).
```
2 [Note: 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]

32.5.4.3.1 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: 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]

Throws: Nothing.

~unique_lock();

Effects: If owns calls pm->unlock().

32.5.4.3.2 Locking

[thread.lock.unique.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:

— operation_not_permitted — if pm is nullptr.
— 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().

Returns: The value returned by the call to try_lock().

Postconditions: owns == res, where res is 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:

— 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);

Preconditions: The supplied Mutex type meets the Cpp17TimedLockable requirements (32.2.5.4).

Effects: As if by pm->try_lock_until(abs_time).

Returns: The value returned by the call to try_lock_until(abs_time).

Postconditions: owns == res, where res is the value returned by the call to try_lock_until(abs_time).

Throws: Any exception thrown by pm->try_lock_until(). 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);

Preconditions: The supplied Mutex type meets the Cpp17TimedLockable requirements (32.2.5.4).

Effects: As if by pm->try_lock_for(rel_time).

Returns: The value returned by the call to try_lock_for(rel_time).

Postconditions: owns == res, where res is the value returned by the call to try_lock_for(rel_time).

Throws: Any exception thrown by pm->try_lock_for(). system_error when an exception is required (32.2.2).

Error conditions:
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 pm is nullptr.

— resource_deadlock_would_occur — if on entry owns is true.

32.5.4.3.3 Modifiers

void swap(unique_lock& u) noexcept;

Effects: Swaps the data members of *this and u.

mutex_type* release() noexcept;

Returns: The previous value of pm.

Postconditions: pm == 0 and owns == false.

template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

Effects: As if by x.swap(y).

32.5.4.3.4 Observers

bool owns_lock() const noexcept;

Returns: owns.

explicit operator bool() const noexcept;

Returns: owns.

mutex_type *mutex() const noexcept;

Returns: pm.

32.5.4.4 Class template shared_lock

namespace std {
    template<class Mutex>
    class shared_lock {
        public:
            using mutex_type = Mutex;

            // 32.5.4.4.1, 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;

§ 32.5.4.4
// 32.5.4.4.2, 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();

// 32.5.4.4.3, modifiers
void swap(shared_lock& u) noexcept;
mutex_type* release() noexcept;

// 32.5.4.4.4, 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
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 (32.5.3.5).

2 [Note: shared_lock<Mutex> meets the Cpp17TimedLockable requirements (32.2.5.4). — end note]

32.5.4.4.1 Constructors, destructor, and assignment [thread.lock.shared.cons]

shared_lock() noexcept;

1 Postconditions: pm == nullptr and owns == false.

explicit shared_lock(mutex_type& m);

2 Preconditions: The calling thread does not own the mutex for any ownership mode.

3 Effects: Calls m.lock_shared().

4 Postconditions: pm == addressof(m) and owns == true.

shared_lock(mutex_type& m, defer_lock_t) noexcept;

5 Postconditions: pm == addressof(m) and owns == false.

shared_lock(mutex_type& m, try_to_lock_t);

6 Preconditions: The calling thread does not own the mutex for any ownership mode.

7 Effects: Calls m.try_lock_shared().

8 Postconditions: pm == addressof(m) and owns == res where res is the value returned by the call to
m.try_lock_shared().

shared_lock(mutex_type& m, adopt_lock_t);

9 Preconditions: The calling thread has shared ownership of the mutex.

10 Postconditions: pm == addressof(m) and owns == true.
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).

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().

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.pm == nullptr and sl.owns == false.

shared_lock& operator=(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 assignment), sl.pm == nullptr and sl.owns == false.

32.5.4.4.2 Locking

void lock();
    Effects: As if by pm->lock_shared().
    Postconditions: owns == true.
    Throses: Any exception thrown by pm->lock_shared(). 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();
    Effects: As if by pm->try_lock_shared().
    Returns: The value returned by the call to pm->try_lock_shared() is the value returned by the call to
             pm->try_lock_shared().
    Throses: Any exception thrown by pm->try_lock_shared(). system_error when an exception is required
             (32.2.2).
    Error conditions:
        (9.1) operation_not_permitted — if pm is nullptr.
        (9.2) 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).
    Returns: The value returned by the call to 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).

Throws: Any exception thrown by pm->try_lock_shared_until(abs_time). system_error when an exception is required (32.2.2).

Error conditions:

(14.1) — operation_not_permitted — if pm is nullptr.
(14.2) — 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).

Returns: The value returned by the call to 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).

Throws: Any exception thrown by pm->try_lock_shared_for(rel_time). system_error when an exception is required (32.2.2).

Error conditions:

(19.1) — operation_not_permitted — if pm is nullptr.
(19.2) — 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:

(23.1) — operation_not_permitted — if on entry owns is false.

32.5.4.4.3 Modifiers

void swap(shared_lock& sl) noexcept;

Effects: Swaps the data members of *this and sl.

mutex_type* release() noexcept;

Returns: The previous value of pm.

Postconditions: pm == nullptr and owns == false.

template<class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;

Effects: As if by x.swap(y).

32.5.4.4.4 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 Generic locking algorithms

```cpp
template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

1. **Preconditions:** Each template parameter type meets the Cpp17Lockable requirements. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

2. **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()`.

3. **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&, L2&, L3&...);
```

4. **Preconditions:** Each template parameter type meets the Cpp17Lockable requirements. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

5. **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: A deadlock avoidance algorithm such as try-and-back-off must 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()`.

32.5.6 Call once

32.5.6.1 Struct `once_flag`

```cpp
namespace std {
  struct once_flag {
    constexpr once_flag() noexcept;
    once_flag(const once_flag&) = delete;
    once_flag& operator=(const once_flag&) = delete;
  }
}
```

1. The class `once_flag` is an opaque data structure that `call_once` uses to initialize data without causing a data race or deadlock.

2. **Synchronization:** The construction of a `once_flag` object is not synchronized.

3. **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.

32.5.6.2 Function `call_once`

```cpp
template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);
```

1. **Mandates:** `is_invocable_v<Callable, Args...>` is true.

2. **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: Passive executions allow other threads to reliably observe the results produced by the earlier returning execution. — end note]

3. **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.

4. **Throws:** `system_error` when an exception is required (32.2.2), or any exception thrown by `func`. 

§ 32.5.6.2 1582
32.6 Condition variables

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.

Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

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.

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.

Condition variable construction and destruction need not be synchronized.

32.6.1 Header `<condition_variable>` synopsis

```cpp
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 };
}
```
32.6.2 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: The supplied lock will be held until the thread exits, and care should be taken to ensure that this does not cause deadlock due to lock ordering issues. After calling notify_all_at_thread_exit it is recommended that the thread should be exited as soon as possible, and that no blocking or time-consuming tasks are run on that thread. —end note]**

5. **[Note: 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]**

32.6.3 Class condition_variable

```
namespace std {
    class condition_variable {
    public:
        condition_variable();
        ~condition_variable();

        condition_variable(const condition_variable&) = delete;
        condition_variable& operator=(const condition_variable&) = delete;

        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
        native_handle_type native_handle();                          // see 32.2.3
    };
}
```

1. The class condition_variable is a standard-layout class (11.2).
condition_variable();

Throws: system_error when an exception is required (32.2.2).

Error conditions:

— resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

~condition_variable();

Preconditions: There is no thread blocked on *this. [Note: 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. The user should take care to ensure that no threads wait 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;

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.

void wait(unique_lock<mutex>& lock);

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() or a call to notify_all(), or spuriously.

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Throws: Nothing.

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:

while (!pred())
    wait(lock);

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

Throws: Any exception thrown by pred.
template<class Clock, class Duration>
    cv_status wait_until(unique_lock<mutex>& lock,
        const chrono::time_point<Clock, Duration>& abs_time);

17  Preconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(17.1) no other thread is waiting on this condition_variable object or
(17.2) 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.

18  Effects:
(18.1) Atomically calls lock.unlock() and blocks on *this.
(18.2) When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
(18.3) 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.
(18.4) If the function exits via an exception, lock.lock() is called prior to exiting the function.

19  Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This
can happen if the re-locking of the mutex throws an exception. — end note]

20  Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

21  Returns: cv_status::timeout if the absolute timeout (32.2.4) specified by abs_time expired, otherwise
    cv_status::no_timeout.

22  Throws: Timeout-related exceptions (32.2.4).

template<class Rep, class Period>
    cv_status wait_for(unique_lock<mutex>& lock,
        const chrono::duration<Rep, Period>& rel_time);

23  Preconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(23.1) no other thread is waiting on this condition_variable object or
(23.2) 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.

24  Effects: Equivalent to:
    return wait_until(lock, chrono::steady_clock::now() + rel_time);

25  Returns: cv_status::timeout if the relative timeout (32.2.4) specified by rel_time expired, otherwise
    cv_status::no_timeout.

26  Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This
can happen if the re-locking of the mutex throws an exception. — end note]

27  Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

28  Throws: Timeout-related exceptions (32.2.4).

template<class Clock, class Duration, class Predicate>
    bool wait_until(unique_lock<mutex>& lock,
        const chrono::time_point<Clock, Duration>& abs_time,
        Predicate pred);

29  Preconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(29.1) no other thread is waiting on this condition_variable object or
(29.2) 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.

30  Effects: Equivalent to:
    while (!pred())
        if (wait_until(lock, abs_time) == cv_status::timeout)
            return pred();
return true;

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: 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.

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

(35.1) no other thread is waiting on this condition_variable object or
(35.2) 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:

return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));

[Note: There is no blocking if pred() is initially true, even if the timeout has already expired. — end note]

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: 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.

32.6.4 Class condition_variable_any

A Lock type shall meet the Cpp17BasicLockable requirements (32.2.5.2). [Note: 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, the user should ensure that any necessary synchronization is 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;

        // 32.6.4.1, 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);
    }

§ 32.6.4 1587
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);

// 32.6.4.2, interruptible waits
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);

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();

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 initial-
ization.

(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: 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. The user should take care to ensure that no threads wait 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;

6

Effects: Unblocks all threads that are blocked waiting for *this.

32.6.4.1 Noninterruptible waits

template<class Lock>
void wait(Lock& lock);

1

Effects:

(1.1) — Atomically calls lock.unlock() and blocks on *this.

(1.2) — When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.

(1.3) — The function will unblock when signaled by a call to notify_one(), a call to notify_all(), or
spuriously.

2

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This
can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: lock is locked by the calling thread.

4

Throws: Nothing.

§ 32.6.4.1 1588
template<class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);

Effects: Equivalent to:
   while (!pred())
       wait(lock);

template<class Lock, class Clock, class Duration>
cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);

Effects:
(6.1) — Atomically calls lock.unlock() and blocks on *this.
(6.2) — When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.
(6.3) — 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.
(6.4) — If the function exits via an exception, lock.lock() is called prior to exiting the function.

Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This
         can happen if the re-locking of the mutex throws an exception. — end note]
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).

template<class Lock, class Rep, class Period>
cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);

Effects: Equivalent to:
   return wait_until(lock, chrono::steady_clock::now() + rel_time);

Returns: cv_status::timeout if the relative timeout (32.2.4) specified by rel_time expired, otherwise
         cv_status::no_timeout.
Remarks: If the function fails to meet the postcondition, terminate() is called (14.6.1). [Note: This
         can happen if the re-locking of the mutex throws an exception. — end note]
Postconditions: lock is locked by the calling thread.
Throws: Timeout-related exceptions (32.2.4).

template<class Lock, class Clock, class Duration, class Predicate>
bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);

Effects: Equivalent to:
   while (!pred())
       if (wait_until(lock, abs_time) == cv_status::timeout)
           return pred();
       return true;
[Note: There is no blocking if pred() is initially true, or if the timeout has already expired. — end
  note]
[Note: 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);

Effects: Equivalent to:
   return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));

32.6.4.2 Interruptible waits [thread.condvarany.intwait]
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);

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();

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether there was a stop request. — end note]

Postconditions: lock is locked by the calling thread.

Remarks: If the function fails to meet the postcondition, terminate is called (14.6.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Throws: Any exception thrown by 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);

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 (cv.wait_until(lock, abs_time) == cv_status::timeout)
    return pred();
}
return pred();

[Note: There is no blocking if pred() is initially true, stoken.stop_requested() was already true or the timeout has already expired. — end note]

[Note: 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]

Postconditions: lock is locked by the calling thread.

Remarks: If the function fails to meet the postcondition, terminate is called (14.6.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Throws: Timeout-related exceptions (32.2.4), or any exception thrown by 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);

Effects: Equivalent to:

return wait_until(lock, std::move(stoken), chrono::steady_clock::now() + rel_time, std::move(pred));

32.7 Semaphore

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.
32.7.1 Header <semaphore> synopsis

namespace std {
    template<ptrdiff_t least_max_value = implementation-defined>
    class counting_semaphore;

    using binary_semaphore = counting_semaphore<1>;
}

32.7.2 Class template counting_semaphore

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
    };
}

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.

least_max_value shall be non-negative; otherwise the program is ill-formed.

Concurrent invocations of the member functions of counting_semaphore, other than its destructor, do not introduce data races.

static constexpr ptrdiff_t max() noexcept;

Returns: The maximum value of counter. This value is greater than or equal to least_max_value.

callcexpr explicit counting_semaphore(ptrdiff_t desired);

Preconditions: desired >= 0 is true, and desired <= max() is true.

Effects: Initializes counter with desired.

Throws: Nothing.

void release(ptrdiff_t update = 1);

Preconditions: update >= 0 is true, and update <= max() - counter is true.

Effects: Atomically execute counter += update. Then, unblocks any threads that are waiting for counter to be greater than zero.

Synchronization: Strongly happens before invocations of try_acquire that observe the result of the effects.

Throws: system_error when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).
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 try_acquire immediately returns. An implementation may fail to decrement counter even if it is positive. [Note: 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_acquire does not consistently return false in the absence of contending semaphore operations.

Returns: true if counter was decremented, otherwise false.

void acquire();

Effects: Repeatedly performs the following steps, in order:

(15.1) Evaluates try_acquire. If the result is true, returns.

(15.2) Blocks on *this until counter is greater than zero.

Throws: system_error when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

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:

(18.1) Evaluates try_acquire(). If the result is true, returns true.

(18.2) Blocks on *this until counter is greater than zero or until the timeout expires. If it is unblocked by the timeout expiring, returns false.

The timeout expires (32.2.4) when the current time is after abs_time (for try_acquire_until) or when at least rel_time has passed from the start of the function (for try_acquire_for).

Throws: Timeout-related exceptions (32.2.4), or system_error when a non-timeout-related exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

32.8 Coordination types

This subclause describes various concepts related to thread coordination, and defines the coordination types latch and barrier. These types facilitate concurrent computation performed by a number of threads.

32.8.1 Latches

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.

32.8.1.1 Header <latch> synopsis

namespace std {
    class latch;
}

32.8.1.2 Class latch

namespace std {
    class latch {
    public:
        static constexpr ptrdiff_t max() noexcept;

        constexpr explicit latch(ptrliff_t expected);
        ~latch();

        latch(const latch&) = delete;
        latch& operator=(const latch&) = delete;

    § 32.8.1.2
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.

Concurrent invocations of the member functions of `latch`, other than its destructor, do not introduce data races.

```
 static constexpr ptrdiff_t max() noexcept;

 constexpr explicit latch(ptrdiff_t expected);

 void count_down(ptrdiff_t update = 1);

 void try_wait() const noexcept;
 void wait() const;
 void arrive_and_wait(ptrdiff_t update = 1);
```

32.8.2 Barriers

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: A barrier is useful for managing repeated tasks that are handled by multiple threads. — end note]

32.8.2.1 Header `<barrier>` synopsis

```
namespace std {
    template<class CompletionFunction = see below>
    class barrier;
}
```
32.8.2.2 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;

        [[nodiscard]] 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.

   1.3 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.

2 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.

3 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.

4 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.

5 CompletionFunction shall meet the Cpp17MoveConstructible (Table 28) and Cpp17Destructible (Table 32)
requirements. is_nothrow_invocable_v<CompletionFunction&> shall be true.

6 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.

7 barrier::arrival_token is an unspecified type, such that it meets the Cpp17MoveConstructible (Table 28),
Cpp17MoveAssignible (Table 30), and Cpp17Destructible (Table 32) requirements.
static constexpr ptrdiff_t max() noexcept;

Returns: The maximum expected count that the implementation supports.

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: If expected is 0 this object can only be destroyed. — end note]

Throws: Any exception thrown by CompletionFunction's move constructor.

[[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 (32.5.3.2).

[Note: This call can cause the completion step for the current phase to start. — end note]

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: If arrival is associated with the
synchronization point for a previous phase, the call returns immediately. — end note]

Throws: system_error when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

void arrive_and_wait();

Effects: Equivalent to: wait(arrive()).

void arrive_and_drop();

Preconditions: The expected count for the current barrier phase is greater than zero.

Effects: Decrement the initial expected count for all subsequent phases by one. Then decrements the
expected count for the current phase by one.

Synchronization: The call to arrive_and_drop strongly happens before the start of the phase completion
step for the current phase.

Throws: system_error when an exception is required (32.2.2).

Error conditions: Any of the error conditions allowed for mutex types (32.5.3.2).

[Note: This call can cause the completion step for the current phase to start. — end note]

32.9 Futures [futures]

32.9.1 Overview [futures.overview]

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. [Note: These components are not
restricted to multi-threaded programs but can be useful in single-threaded programs as well. — end note]
32.9.2 Header <future> synopsis

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<> 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);
}

[Note: 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

```cpp
const error_category& future_category() noexcept;
```

1 Returns: A reference to an object of a type derived from class error_category.

2 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".

```cpp
error_code make_error_code(future_errc e) noexcept;
```

3 Returns: error_code(static_cast<int>(e), future_category()).

```cpp
error_condition make_error_condition(future_errc e) noexcept;
```

4 Returns: error_condition(static_cast<int>(e), future_category()).

### 32.9.4 Class future_error

```cpp
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
    };
}
```

1 explicit future_error(future_errc e);

   Effects: Initializes ec_ with make_error_code(e).

```cpp
const error_code& code() const noexcept;
```

2 Returns: ec_.

```cpp
const char* what() const noexcept;
```

3 Returns: An NTBS incorporating code().message().

### 32.9.5 Shared 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: Futures, promises, and tasks defined in this clause reference such shared state. —end note]

2 [Note: The result can be any kind of object including a function to compute that result, as used by async when policy is launch::deferred. —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.

4 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: 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.

5 When an asynchronous return object or an asynchronous provider is said to release its shared state, it means:
if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and

the return object or provider gives up its reference to its shared state; and

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:

— first, the provider marks its shared state as ready; and

— 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:

— first, if that state is not ready, the provider

— stores an exception object of type future_error with an error condition of broken_promise within its shared state; and then

— makes its shared state ready;

— 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 (6.7.5.2) is sequenced before making that shared state ready.

Access to the result of the same shared state may conflict (6.9.2). [Note: 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 (16.5.5.10). 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]
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.

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.

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).

4. Effects: Creates a shared state. The second constructor uses the allocator a to allocate memory for the shared state.

5. Effects: Transfers ownership of the shared state of rhs (if any) to the newly-constructed object.

6. Postconditions: rhs has no shared state.

7. Effects: Abandons any shared state (32.9.5).

8. Effects: Abandons any shared state (32.9.5) and then as if promise(std::move(rhs)).swap(*this).

9. Returns: *this.

10. Effects: Exchanges the shared state of *this and other.

11. 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.

12. Returns: A future<R> object with the same shared state as *this.

13. 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: Such calls need not synchronize with each other. — end note]

14. 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.

15. 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.

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.

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);

Preconditions: p is not null.

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.

Throws: future_error if an error condition occurs.

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;

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: 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: It is valid to move from a future object for which `valid() == false`. — end note] [Note: Implementations should detect this case and throw an object of type `future_error` with an error condition of `future_errc::no_state`. — end note]

```cpp
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;

        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 `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:** `valid() == 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:**

(8.1) — `valid()` returns the same value as `rhs.valid()` prior to the constructor invocation.

(8.2) — `rhs.valid() == false`.

`~future();`

**Effects:**

(9.1) — Releases any shared state (32.9.5);
future\& operator=(future\&\& rhs) noexcept;

Effects:

(10.1) Releases any shared state (32.9.5).
(10.2) move assigns the contents of rhs to *this.

Postconditions:

(11.1) valid() returns the same value as rhs.valid() prior to the assignment.
(11.2) rhs.valid() == false.

shared_future<R> share() noexcept;

Returns: shared_future<R>(std::move(*this)).

Postconditions: valid() == false.

\[\text{Note: As described above, the template and its two required specializations differ only in the return type and return value of the member function get. — end note}\]

future::get();
future<R&>::get();
void future<void>::get();

Effects:

(15.1) wait() \(\)s until the shared state is ready, then retrieves the value stored in the shared state;
(15.2) releases any shared state (32.9.5).

Returns:

(16.1) future::get() returns the value \(v\) stored in the object’s shared state as std::move(\(v\)).
(16.2) future<R&>::get() returns the reference stored as value in the object’s shared state.
(16.3) future<void>::get() returns nothing.

Throws: The stored exception, if an exception was stored in the shared state.

Postconditions: valid() == false.

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:
— **future_status::deferred** if the shared state contains a deferred function.

— **future_status::ready** if the shared state is ready.

— **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`  
[`futures.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.

2 [Note: Member functions of `shared_future` do not synchronize with themselves, but they synchronize with the shared state. — end note]

3 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: It is valid to copy or move from a `shared_future` object for which `valid()` is false. — end note]

[Note: Implementations should detect this case and throw an object of type `future_error` with an error condition of `future_errc::no_state`. — end note]

```cpp
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;
        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;
    
    private:
        // implementation
    
    }; // namespace std
```

4 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()`.

§ 32.9.8
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:

(10.1) valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
(10.2) rhs.valid() == false.

~shared_future();

Effects:

(11.1) Releasing any shared state (32.9.5);
(11.2) destroys *this.

shared_future& operator=(shared_future&& rhs) noexcept;

Effects:

(12.1) Releasing any shared state (32.9.5);
(12.2) move assigns the contents of rhs to *this.

Postconditions:

(13.1) valid() returns the same value as rhs.valid() returned prior to the assignment.
(13.2) rhs.valid() == false.

shared_future& operator=(const shared_future& rhs) noexcept;

Effects:

(14.1) Releasing any shared state (32.9.5);
(14.2) assigns the contents of rhs to *this. [Note: 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: 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: Access to a value object stored in the shared state is unsynchronized, so programmers should apply only those operations on R that do not 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: 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

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>
[[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);

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<Args>...>, 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 second function 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))...) (20.14.3, 32.4.2.2) 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. [Note: 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. —end note]

(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: If a future obtained from `async` is moved outside the local scope, other code that uses the future should be aware that the future’s destructor can block for the shared state to become ready. —end note]

Synchronization: Regardless of the provided `policy` argument,

(6.1) the invocation of `async` synchronizes with (6.9.2) the invocation of `f`. [Note: This statement applies even when the corresponding `future` object is moved to another thread. —end note] ; and

(6.2) the completion of the function `f` is sequenced before (6.9.2) the shared state is made ready. [Note: `f` might not be called at all, so its completion might never happen. —end note]

If the implementation chooses the `launch::async` policy,

(6.3) 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 (32.4.2.5);

(6.4) 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:

(8.1) `resource_unavailable_try_again` — if `policy == launch::async` and the system is unable to start a new thread.

[Example:

```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: Line #1 might not result in concurrency because the `async` call uses the default policy, which may 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]

1 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.

2 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.

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();

    // execution
    void operator()(ArgTypes...);
    void make_ready_at_thread_exit(ArgTypes...);
    void reset();
  };

  template<class R, class... ArgTypes>
  void swap(packaged_task<R(ArgTypes...)>& x, packaged_task<R(ArgTypes...)>& y) noexcept;
};

32.9.10.1  Member functions  [futures.task.members]

packaged_task() noexcept;

1 Effects: The object has no shared state and no stored task.

template<class F>
packaged_task(F&& f);

2 Constraints: remove_cvref_t<F> is not the same type as packaged_task<R(ArgTypes...)>.  
3 Mandates: is_invocable_r_v<R, F, ArgTypes...> is true.
4 Preconditions: Invoking a copy of f behaves the same as invoking f.
5 Effects: Constructs a new packaged_task object with a shared state and initializes the object’s stored
  task with std::forward<F>(f).
6 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.

§ 32.9.10.1  1607
packaged_task(packaged_task&& rhs) noexcept;
7 Effects: Transfers ownership of rhs’s shared state to *this, leaving rhs with no shared state. Moves the stored task from rhs to *this.
8 Postconditions: rhs has no shared state.

packaged_task& operator=(packaged_task&& rhs) noexcept;
9 Effects:

(9.1) Releases any shared state (32.9.5);
(9.2) calls packaged_task(std::move(rhs)).swap(*this).

~packaged_task();
10 Effects: Abandons any shared state (32.9.5).

void swap(packaged_task& other) noexcept;
11 Effects: Exchanges the shared states and stored tasks of *this and other.
12 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;
13 Returns: true only if *this has a shared state.

future<R> get_future();
14 Returns: A future object that shares the same shared state as *this.
15 Synchronization: Calls to this function do not introduce data races (6.9.2) with calls to operator() or make_ready_at_thread_exit. [Note: Such calls need not synchronize with each other. — end note]
16 Throws: A future_error object if an error occurs.
17 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.

void operator()(ArgTypes... args);
18 Effects: As if by INVOKE<R>(f, t_1, t_2, ..., t_N) (20.14.3), where f is the stored task of *this and t_1, t_2, ..., t_N 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.
19 Throws: A future_error exception object if there is no shared state or the stored task has already been invoked.
20 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);
21 Effects: As if by INVOKE<R>(f, t_1, t_2, ..., t_N) (20.14.3), where f is the stored task and t_1, t_2, ..., t_N 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.
22 Throws: future_error if an error condition occurs.
23 Error conditions:
promise_already_satisfied if the stored task has already been invoked.

no_state if *this has no shared state.

```cpp
void reset();
```

Effects: As if *this = packaged_task(std::move(f)), where f is the task stored in *this. [Note: This constructs a new shared state for *this. The old state is abandoned (32.9.5). — end note]

Throws:

- bad_alloc if memory for the new shared state could not be allocated.
- any exception thrown by the move constructor of the task stored in the shared state.
- future_error with an error condition of no_state if *this has no shared state.

32.9.10.2 Globals

```cpp
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

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) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

A.1 Keywords

New context-dependent keywords are introduced into a program by typedef (9.2.3), namespace (9.8.1), class (Clause 11), enumeration (9.7.1), and template (Clause 13) declarations.

A.2 Lexical conventions

§ A.2 1610
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
  {  } [ ] ( )
  <:  :>
  ?:  :
  ! =
  ==
  <<
  <:
  or  xor  not  bitand  bitor  compl
  and  eq  or_eq  xor_eq  not_eq

§ A.2
literal:
  integer-literal
  character-literal
  floating-point-literal
  string-literal
  boolean-literal
  pointer-literal
  user-defined-literal

integer-literal:
  binary-literal integer-suffix\textsubscript{opt}
  octal-literal integer-suffix\textsubscript{opt}
  decimal-literal integer-suffix\textsubscript{opt}
  hexadecimal-literal integer-suffix\textsubscript{opt}

binary-literal:
  \texttt{0b} binary-digit
  \texttt{0B} binary-digit
  binary-literal '\,'\textsubscript{opt} binary-digit

octal-literal:
  \texttt{0}
  octal-literal '\,'\textsubscript{opt} octal-digit

decimal-literal:
  nonzero-digit
decimal-literal '\,'\textsubscript{opt} digit

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

binary-digit: one of
  \texttt{0} \texttt{1}

octal-digit: one of
  \texttt{0} \texttt{1} \texttt{2} \texttt{3} \texttt{4} \texttt{5} \texttt{6} \texttt{7}

nonzero-digit: one of
  \texttt{1} \texttt{2} \texttt{3} \texttt{4} \texttt{5} \texttt{6} \texttt{7} \texttt{8} \texttt{9}

hexadecimal-prefix: one of
  \texttt{0x} \texttt{0X}

hexadecimal-digit-sequence:
  hexadecimal-digit
  hexadecimal-digit-sequence '\,'\textsubscript{opt} hexadecimal-digit

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

integer-suffix:
  unsigned-suffix long-suffix\textsubscript{opt}
  unsigned-suffix long-long-suffix\textsubscript{opt}
  long-suffix unsigned-suffix\textsubscript{opt}
  long-long-suffix unsigned-suffix\textsubscript{opt}

unsigned-suffix: one of
  \texttt{u} \texttt{U}

long-suffix: one of
  \texttt{L}

long-long-suffix: one of
  \texttt{LL}

character-literal:
  encoding-prefix\textsubscript{opt} '\,' c-char-sequence '\,'

encoding-prefix: one of
  \texttt{u8} \texttt{u} \texttt{U} \texttt{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
    \' \" \? \\\n\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-sequenceopt ( r-char-sequenceopt ) d-char-sequenceopt 

r-char-sequence:
    d-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-partopt 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.3 Basics

translation-unit:
    declaration-seqopt
global-module-fragmentopt module-declaration declaration-seqopt private-module-fragmentopt

A.4 Expressions
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 ) declspecifier-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-list_opt 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

simple-requirement:
  expression ;

type-requirement:
  typename nested-name-specifier_opt type-name ;

compound-requirement:
  { expression } noexcept_opt return-type-requirement_opt ;

return-type-requirement:
  -> type-constraint

nested-requirement:
  requires constraint-expression ;

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
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}
  noptr-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.5 Statements

§ A.5 1618
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 ; 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.6 Declarations

[gram.dcl]
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
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-seqopt
  type-specifier type-specifier-seq

defining-type-specifier:
  type-specifier
  class-specifier
  enum-specifier

defining-type-specifier-seq:
  defining-type-specifier attribute-specifier-seqopt
  defining-type-specifier defining-type-specifier-seq

simple-type-specifier:
  nested-name-specifieropt type-name
  nested-name-specifier template simple-template-id
  decltype-specifier
  placeholder-type-specifier
  nested-name-specifieropt 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-seqopt nested-name-specifieropt identifier
  class-key simple-template-id
  class-key nested-name-specifier templateopt simple-template-id
  elaborated-enum-specifier

elaborated-enum-specifier:
  enum nested-name-specifieropt identifier

dcltype-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-expression opt ] 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 ;

enum-name:
  identifier

enum-specifier:
  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
  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

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 , using-declarator . . .\opt

using-declarator:
  typename\opt nested-name-specifier unqualified-id
asm-declaration:
  attribute-specifier-seq\_opt asm ( string-literal ) ;

linkage-specification:
  extern string-literal { declaration-seq\_opt }
  extern string-literal declaration

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-namespace :

attribute-list:
  attribute\_opt
  attribute-list , attribute\_opt
  attribute ... attribute-list , attribute ...

attribute:
  attribute-token attribute-argument-clause\_opt

attribute-token:
  identifier
  attribute-scoped-token

attribute-scoped-token:
  attribute-namespace :: 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

A.7 Modules  [gram.module]

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.8 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
conversion-function-id:
  operator conversion-type-id

conversion-type-id:
  type-specifier-seq conversion-declarator opt

conversion-declarator:
  ptr-operator conversion-declarator opt

base-clause:
  : base-specifier-list

base-specifier-list:
  base-specifier ... opt
  base-specifier-list , base-specifier ... opt

base-specifier:
  attribute-specifier-seq opt class-or-decltype
  attribute-specifier-seq opt virtual access-specifier opt class-or-decltype
  attribute-specifier-seq opt access-specifier virtual opt class-or-decltype

class-or-decltype:
  nested-name-specifier opt type-name
  nested-name-specifier template simple-template-id
dectype-specifier

access-specifier:
  private
  protected
  public

cTOR-initializer:
  : mem-initializer-list

mem-initializer-list:
  mem-initializer ... opt
  mem-initializer-list , mem-initializer ... opt

mem-initializer:
  mem-initializer-id ( expression-list opt )
  mem-initializer-id braced-init-list

mem-initializer-id:
  class-or-decltype
  identifier

A.9 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.10 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 identifier opt
  type-parameter-key identifier opt = type-id
  type-constraint ...opt identifier opt
  type-constraint identifier opt = type-id
  template-head type-parameter-key ...opt identifier opt
  template-head type-parameter-key identifier opt = id-expression

type-parameter-key:
  class
  typename

type-constraint:
  nested-name-specifier opt concept-name
  nested-name-specifier opt concept-name < template-argument-list opt >

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

constraint-expression:
  logical-or-expression

deduction-guide:
  explicit-specifier opt 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 template opt simple-template-id

explicit-instantiation:
  extern opt template declaration
A.11 Exception handling

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.12 Preprocessing directives

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
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 white-space

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  (informative)
Implementation quantities

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.

---

1. Nesting levels of compound statements (8.4), iteration control structures (8.6), and selection control structures (8.5) [256].
2. Nesting levels of conditional inclusion (15.2) [256].
3. Pointer (9.3.3.1), array (9.3.3.4), and function (9.3.3.5) declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].
4. Nesting levels of parenthesized expressions (7.5.3) within a full-expression [256].
5. Number of characters in an internal identifier (5.10) or macro name (15.6) [1024].
6. Number of characters in an external identifier (5.10, 6.6) [1024].
7. External identifiers (6.6) in one translation unit [65536].
8. Identifiers with block scope declared in one block (6.4.3) [1024].
9. Structured bindings (9.6) introduced in one declaration [256].
10. Macro identifiers (15.6) simultaneously defined in one translation unit [65536].
11. Parameters in one function definition (9.5.1) [256].
12. Arguments in one function call (7.6.1.2) [256].
13. Parameters in one macro definition (15.6) [256].
14. Arguments in one macro invocation (15.6) [256].
15. Characters in one logical source line (5.2) [65536].
16. Characters in a string-literal (5.13.5) (after concatenation (5.2)) [65536].
17. Size of an object (6.7.2) [262144].
18. Nesting levels for #include files (15.3) [256].
19. Case labels for a switch statement (8.5.2) (excluding those for any nested switch statements) [16384].
20. Non-static data members (including inherited ones) in a single class (11.4) [16384].
21. Lambda-captures in one lambda-expression (7.5.5.2) [256].
22. Enumeration constants in a single enumeration (9.7.1) [4096].
23. Levels of nested class definitions (11.4.10) in a single member-specification [256].
24. Functions registered by atexit() (17.5) [32].
25. Functions registered by at_quick_exit() (17.5) [32].
26. Direct and indirect base classes (11.7) [16384].
27. Direct base classes for a single class (11.7) [1024].
28. Class members declared in a single member-specification (including member functions) (11.4) [4096].
29. Final overriding virtual functions in a class, accessible or not (11.7.2) [16384].
30. Direct and indirect virtual bases of a class (11.7.1) [1024].
(2.31) — Static data members of a class (11.4.8.2) [1024].
(2.32) — Friend declarations in a class (11.9.3) [4096].
(2.33) — Access control declarations in a class (11.9.1) [4096].
(2.34) — Member initializers in a constructor definition (11.10.2) [6144].
(2.35) — \textit{initializer-clauses} in one \textit{braced-init-list} (9.4) [16384].
(2.36) — Scope qualifications of one identifier (7.5.4.2) [256].
(2.37) — Nested \textit{linkage-specifications} (9.11) [1024].
(2.38) — Recursive constexpr function invocations (9.2.5) [512].
(2.39) — Full-expressions evaluated within a core constant expression (7.7) [1048576].
(2.40) — Template parameters in a template declaration (13.2) [1024].
(2.41) — Recursively nested template instantiations (13.9.1), including substitution during template argument deduction (13.10.2) [1024].
(2.42) — Handlers per try block (14.4) [256].
(2.43) — Number of placeholders (20.14.14.4) [10].
Annex C  (informative)
Compatibility 

C.1  C++ and ISO C++ 2017

This subclause lists the differences between C++ and ISO C++ 2017 (ISO/IEC 14882:2017, Programming Languages — C++), by the chapters of this document.

C.1.1  Clause 5: lexical conventions

1  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 International Standard. [Example:

```c++
class module {}
module m1;  // was variable declaration; now module-declaration
module *m2;  // variable declaration

class import {}
import j1;   // was variable declaration; now import-declaration
::import j2; // variable declaration
```
—end example]

2  Affected subclause: 5.8
Change: header-name tokens are formed in more contexts.
Rationale: Required for new features.
Effect on original feature: When the identifier import is followed by a < character, a header-name token may be formed. [Example:

```c++
template<typename> class import {}
import<int> f();  // ill-formed; previously well-formed
::import<int> g();  // OK
```
—end example]

3  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 (13.7.8).
(3.3) — The constexpr keyword is added to declare immediate functions (9.2.5).
(3.4) — The constinit keyword is added to prevent unintended dynamic initialization (9.2.6).
(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 International Standard.

4  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 International Standard:
namespace N {
  struct X {}
  bool operator<=(X, X);
  template<bool(X, X)> struct Y {}
  Y<operator<=> y; // ill-formed; previously well-formed
}

5 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 International Standard.

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

template<typename> struct ct;
template<> struct ct<char> {
  using type = char;
};
ct<decltype(u8'c')>::type x; // ill-formed; previously well-formed.

C.1.2 Clause 6: basics

1 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 International Standard. [Example:

```cpp
int f() {
  int a = 123;
  using T = int;
  a.-T();
  return a; // undefined behavior; previously returned 123
}
```
—end example]

2 Affected subclause: 6.9.2.1
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 memory_order_release atomic store is followed by a memory_order_acquire load no longer provides any “happens before” guarantees, even in the absence of intervening stores by another thread.

C.1.3 Clause 7: expressions

1 Affected subclause: 7.5.5.2
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.1.4 Clause 9: declarations

1 Affected subclause: 9.2.3
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 International Standard.

    typedef struct {
        void f() {} // ill-formed; previously well-formed
    } S;

2 Affected subclause: 9.3.3.6
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 International Standard, 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: 9.4.1
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 International Standard.

    struct A { // not an aggregate; previously an aggregate
        A() = delete;
    };

    struct B { // not an aggregate; previously an aggregate
        B() = default;
        int i = 0;
    };

    struct C { // not an aggregate; previously an aggregate
        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;
    };

    Y y(x()); // copy constructor call; previously aggregate-initialization

4 Affected subclause: 9.4.4
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 International Standard. For example:

```cpp
bool y[] = { "bc" };  // ill-formed; previously well-formed
```

### C.1.5 Clause 11: classes

[Affected subclauses: 11.4.4 and 11.4.7.2]

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 International Standard. 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
};
```

[Affected subclauses: 11.4.4 and 11.4.6]

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 International Standard. 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
};
```

[Affected subclause: 11.10.5]

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 International Standard. 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 &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.1.6 Clause 12: overloading

Affected subclause: 12.4.1.2

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.1.7 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.1.8 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 International Standard. It should simply be replaced with noexcept for no change of meaning since C++ 2017. [Note: There is no way to write a function declaration that is non-throwing in this International Standard 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.1.8

1637
C.1.9  **Clause 16: library introduction**

1 AFFECTED SUBCLAUSE: 16.5.1.2  
CHANGE: New headers.  
RATIONALE: New functionality.  
**Effect on original feature:** The following C++ headers are new: `<barrier>` (32.8.2.1), `<bit>` (26.5.2), `<charconv>` (20.19.1), `<compare>` (17.11.1), `<concepts>` (18.3), `<coroutine>` (17.12.1), `<format>` (20.20.1), `<latch>` (32.8.1.1), `<numbers>` (26.9.1), `<ranges>` (24.2), `<semaphore>` (32.7.1), `<source_location>` (17.8.1), `<span>` (22.7.2), `<stop_token>` (32.3.2), `<syncstream>` (29.10.1), and `<version>` (17.3.1). Valid C++ 2017 code that `#includes` headers with these names may be invalid in this International Standard.

2 AFFECTED SUBCLAUSE: 16.5.1.2  
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 `#includes` any of the following headers may fail to compile: `<ccomplex>`, `<ciso646>`, `<cstdalign>`, `<cstdbool>`, and `<ctgmath>`. To retain the same behavior:

(2.1) — a `#include` of `<ccomplex>` can be replaced by a `#include` of `<complex>` (26.4.1),

(2.2) — a `#include` of `<ctgmath>` can be replaced by a `#include` of `<cmath>` (26.8.1) and a `#include` of `<complex>`, and

(2.3) — a `#include` of `<ciso646>`, `<cstdalign>`, or `<cstdbool>` can simply be removed.

C.1.10  **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 International Standard. 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.1.11  **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.1.12  **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.1.13  **Clause 29: input/output library**

1 AFFECTED SUBCLAUSE: 29.7.4.2.3  
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 International Standard:

```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: 29.7.5.2.4  
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: 29.7.5.2.4

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 << u"text"; // previously formatted the string as a pointer value;
// now ill-formed
std::cout << u"X"; // previously formatted the character as an integer value;
// now ill-formed
```

4 Affected subclause: 29.11.7

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 International Standard.

```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.1.14 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.

6 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.

7 Affected subclause: D.13
   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

This subclause 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.2.1 Clause 5: lexical conventions

1 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 International Standard. 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.

2 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 International Standard. Specifically, character sequences like `0p+0` and `0el_p+0` are three separate tokens in C++ 2014, but one single token in this International Standard. For example:

```c++
#define F(a) b ## a
int b0p = F(0p+0);  // ill-formed; equivalent to "int b0p = b0p + 0;" in C++ 2014
```

C.2.2 Clause 7: expressions

1 Affected subclauses: 7.6.1.5 and 7.6.2.2
   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 International Standard. Note that this might occur when the lvalue has a type given by a template parameter.

2 Affected subclauses: 7.6.2.7 and 7.6.2.8
   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 International Standard, ::operator new(std::size_t, std::align_val_t) is used instead.
C.2.3 Clause 9: declarations

1 Affected subclause: 9.2.1
Change: Removal of register storage-class-specifier.
Rationale: Enable repurposing of deprecated keyword in future revisions of this International Standard.
Effect on original feature: A valid C++ 2014 declaration utilizing the register storage-class-specifier is ill-formed in this International Standard. The specifier can simply be removed to retain the original meaning.

2 Affected subclause: 9.2.8.5
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 International Standard. 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: 9.3.3.5
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 International Standard. For example:

```cpp
void g1() noexcept;
void g2();
template<class T> int f(T *, T *);
int x = f(g1, g2);  // ill-formed; previously well-formed
```

4 Affected subclause: 9.4.1
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 International Standard; 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.2.4 Clause 11: classes

1 Affected subclause: 11.10.3
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.2.5 Clause 13: templates

Affected subclause: 13.10.2.5

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 International Standard. For example:

```cpp
template <int N> struct A;
template <typename T, T N> int foo(A<N> *) = delete;
void foo(void *p);
void bar(A<0> *p) {
  foo(p);
  // ill-formed; previously well-formed
}
```

C.2.6 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 International Standard 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, will be rejected as ill-formed in this International Standard. Violating a non-throwing dynamic exception specification will call `terminate` rather than `unexpected` and might not perform stack unwinding prior to such a call.

C.2.7 Clause 16: library introduction

Affected subclause: 16.5.1.2

Change: New headers.

Rationale: New functionality.

Effect on original feature: The following C++ headers are new: `<any>` (20.8.1), `<charconv>` (20.19.1), `<execution>` (20.18.2), `<filesystem>` (29.11.5), `<memory_resource>` (20.12.1), `<optional>` (20.6.2), `<string_view>` (21.4.1), and `<variant>` (20.7.2). Valid C++ 2014 code that `#includes` headers with these names may be invalid in this International Standard.

Affected subclause: 16.5.4.2.3

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 International Standard.

C.2.8 Clause 20: general utilities library

Affected subclause: 20.14.16

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 International Standard. 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 International Standard.

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 International Standard. 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.2.9 Clause 21: strings library

Affected subclause: 21.3.2
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 for this International Standard.

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 International Standard.

```cpp
int f(char *) = delete;
int f(const char *);
string s;
int x = f(s.data()); // ill-formed; previously well-formed
```

C.2.10 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 International Standard:

```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.2.11 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 International Standard.

2 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 International Standard.

C.3 C++ and ISO C++ 2011

This subclause 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.3.1 Clause 5: lexical conventions

Affected subclause: 5.9
Change: pp-number can contain one or more single quotes.
Rationale: Necessary to enable single quotes as digit separators.

Effect on original feature: Valid C++ 2011 code may fail to compile or may change meaning in this International Standard. For example, the following code is valid both in C++ 2011 and in this International
Standard, but the macro invocation produces different outcomes because the single quotes delimit a character-literal in C++ 2011, whereas they are digit separators in this International Standard:

```c++
#define M(x, ...) __VA_ARGS__
int x[2] = { M(1'2,3'4, 5) };
// int x[2] = { 3'4, 5 }; — this International Standard
```

C.3.2 Clause 6: basics

Affected subclause: 6.7.5.4.2

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:

```c++
void* operator new(std::size_t, std::size_t);
void operator delete(void*, std::size_t) noexcept;
```

In this International Standard, however, the declaration of `operator delete` might match a predefined usual (non-placement) `operator delete` (6.7.5.4). If so, the program is ill-formed, as it was for class member allocation functions and deallocation functions (7.6.2.7).

C.3.3 Clause 7: expressions

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 (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) 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 International Standard:

```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 International Standard, it returns 2.

```c++
sizeof(true ? "" : throw 0)
```

In C++ 2011, the expression yields `sizeof(const char*)`. In this International Standard, it yields `sizeof(const char[1])`.

C.3.4 Clause 9: declarations

Affected subclause: 9.2.5

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 International Standard. For example, the following code is valid in C++ 2011 but invalid in this International Standard because it declares the same member function twice with different return types:

```c++
struct S {
    constexpr const int &f();
    int &f();
};
```

Affected subclause: 9.4.1

Change: Classes with default member initializers can be aggregates.

Rationale: Necessary to allow default member initializers to be used by aggregate initialization.

§ C.3.4
Effect on original feature: Valid C++ 2011 code may fail to compile or may change meaning in this International Standard. For example:

```cpp
struct S { // Aggregate in C++ 2014 onwards.
    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 International Standard
```

C.3.5 Clause 16: library introduction

1 Affected subclause: 16.5.1.2
Change: New header.
Rationale: New functionality.
Effect on original feature: The C++ header `<shared_mutex>` (32.5.2) is new. Valid C++ 2011 code that includes a header with that name may be invalid in this International Standard.

C.3.6 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 International Standard.

C.4 C++ and ISO C++ 2003

1 This subclause 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.4.1 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 International Standard. 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 "abcdef", now "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 International Standard. 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 International Standard, #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`, `decltype`, `noexcept`, `nullptr`, `static_assert`, and `thread_local`. Valid C++ 2003 code using these identifiers is invalid in this International Standard.

§ C.4.1 1645
4 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.4.2 Clause 7: expressions [diff.cpp03.expr]
1 Affected subclause: 7.3.11
   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 International Standard. For example:
   
   void f(void *); // #1
   void f(...); // #2
   template<int N> void g() {
      f(0*N); // calls #2; used to call #1
   }

2 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 International Standard always rounds the result toward 0.

3 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 in this International Standard. For example, the following code is valid in C++ 2003 but invalid in this International Standard because double to int is a narrowing conversion:

   int x[] = { 2.0 };

C.4.3 Clause 9: declarations [diff.cpp03.dcl.dcl]
1 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 International Standard. In this International Standard, auto indicates that the type of a variable is to be deduced from its initializer expression.

2 Affected subclause: 9.4.4
   Change: Narrowing restrictions in aggregate initializers.
   Rationale: Catches bugs.
   Effect on original feature: Valid C++ 2003 code may fail to compile in this International Standard. For example, the following code is valid in C++ 2003 but invalid in this International Standard because double to int is a narrowing conversion:

   int x[] = { 2.0 };

C.4.4 Clause 11: classes [diff.cpp03.class]
1 Affected subclauses: 11.4.4.1, 11.4.6, 11.4.4.2, and 11.4.5
   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.

2 Affected subclause: 11.4.6
   Change: User-declared destructors have an implicit exception specification.

§ C.4.4 1646
Rationale: Clarification of destructor requirements.

Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. In particular, destructors that throw exceptions will call `std::terminate` (without calling `std::unexpected`) if their exception specification is non-throwing.

C.4.5 Clause 13: templates [diff.cpp03.temp]

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 International Standard.

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 International Standard because ">>" closes two templates.

   ```cpp
   template <class T> struct X { }
   template <int N> struct Y { }
   X< Y< 1 >> 2 >> x;
   ```

3 Affected subclause: 13.8.4.2
   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 this International Standard.

C.4.6 Clause 16: library introduction [diff.cpp03.library]

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 this International Standard may fail to compile or produce different results in this International Standard. A comprehensive list of identifiers used by the C++ standard library can be found in the Index of Library Names in this International Standard.

2 Affected subclause: 16.5.1.2
   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.20.1), `<condition_variable>` (32.6.1), `<forward_list>` (22.3.4), `<future>` (32.9.2), `<initializer_list>` (17.10.1), `<mutex>` (32.5.1), `<random>` (26.6.1), `<ratio>` (20.16.2), `<regex>` (30.4), `<scoped_allocator>` (20.13.1), `<system_error>` (19.5.1), `<thread>` (32.4.1), `<tuple>` (20.5.2), `<typeindex>` (20.17.1), `<type_traits>` (20.15.2), `<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>` (17.4.1), and `<cuchar>` (21.5.5). Valid C++ 2003 code that includes headers with these names may be invalid in this International Standard.

3 Affected subclause: 16.5.3.2
   Effect on original feature: Function `swap` moved to a different header
   Rationale: Remove dependency on `<algorithm>` (25.4) for `swap`.

4 Affected subclause: 16.5.4.2.2
   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 International Standard.

Affected subclause: 16.5.5.3
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 International Standard.

C.4.7 Clause 17: language support library

Affected subclause: 17.6.2.1
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 International Standard. Valid C++ 2003 code that replaces the global replaceable `operator new` is ill-formed in this International Standard, because the exception specification of `throw(std::bad_alloc)` was removed.

C.4.8 Clause 19: diagnostics library

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 International Standard.

C.4.9 Clause 20: general utilities library

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 International Standard.

C.4.10 Clause 21: strings library

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 for this International Standard.
Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard.

Affected subclause: 21.3.2.1
Change: Loosen `basic_string` invalidation rules.
Rationale: Allow small-string optimization.
Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. Some `const` member functions, such as `data` and `c_str`, no longer invalidate iterators.

C.4.11 Clause 22: containers library

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 International Standard. Adjusting containers such as std::list to the stricter requirements may require incompatible changes.

2 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:

(2.1) not all containers provide size(); use empty() instead of size() == 0;
(2.2) not all containers are empty after construction (array);
(2.3) not all containers have constant complexity for swap() (array).

3 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.

4 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:

(4.1) erase(iter) for set, multiset, map, multimap
(4.2) erase(begin, end) for set, multiset, map, multimap
(4.3) insert(pos, num, val) for vector, deque, list, forward_list
(4.4) 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 International Standard.

5 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:

(5.1) insert(iter, val) for vector, deque, list, set, multiset, map, multimap
(5.2) insert(pos, beg, end) for vector, deque, list, forward_list
(5.3) erase(begin, end) for set, multiset, map, multimap
(5.4) all forms of list::splice
(5.5) all forms of list::merge

Valid C++ 2003 code that uses these functions may fail to compile with this International Standard.

6 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 International Standard.

C.4.12 Clause 25: algorithms library [diff.cpp03.algorithms]

1 Affected subclause: 25.1
Change: Result state of inputs after application of some algorithms.
Rationale: Required by new feature.
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 International Standard. 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.4.13 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 International Standard.

C.4.14 Clause 29: input/output library

Affected subclauses: 29.7.4.1.3, 29.7.5.1.3, 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 International Standard. 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: 29.5.3.1.1
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 International Standard.

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 International Standard. For example:

```
#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

This subclause lists the differences between C++ and ISO C, in addition to those listed above, by the chapters of this document.

C.5.1 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:
int function( int i );
int function( char 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 `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.

### Clause 6: basics

1. **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.

2. **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 9.2.8]

  **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.5.3 **Clause 7: expressions**

1 **Affected subclause:** 7.3.11

  **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:** 7.6.1.2

  **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:** 7.6.1.5 and 7.6.2.2

  **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.9.4)) is ill-formed in this International Standard.

Affected subclauses: 7.6.2.4 and 7.6.3
Change: Types must be defined in declarations, not in expressions.
In C, a sizeof expression or cast expression may define a new type. For example,
\[ p = (void*) (\text{struct } x \{ \text{int } i; \} \ast)0; \]
defines a new type, struct x.
Rationale: 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.

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.
Rationale: 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,
\[ \text{char arr[100];} \\
\text{sizeof(0, arr)} \]
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.5.4 Clause 8: statements

Affected subclauses: 8.5.2 and 8.7.5
Change: It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered).
Rationale: 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.

Affected subclause: 8.7.3
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.
Rationale: 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.5.5 Clause 9: declarations

Affected subclause: 9.2.1
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: 9.2.1

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: 9.2.3

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:

typedef struct name1 { /* ... */ } name1;       // valid C and C++
struct name { /* ... */ };
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:

class name { /* ... */ };
name 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: 9.2.8 [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: 9.2.8

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++:

```
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:** 9.2.8.5
**Change:** The keyword `auto` cannot be used as a storage class specifier.

```c
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:** 9.3.3.5
**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.

```c
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 “obsolete” 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:** 9.3.3.5 [see 7.6.2.4]
**Change:** In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed.

```c
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 “obsolete”.

**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 obsolete.

10 **Affected subclause:** 9.4.1
**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:

```c
struct A { int x, y; }
struct B { struct A a; }
struct A a = {.y = 1, .x = 2}; // valid C, invalid C++
int arr[3] = {{1} = 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: 9.4.2

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 }
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: Syntactic 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 }
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 affects 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.5.6 Clause 11: classes

Affected subclause: 11.3 [see also 9.2.3]

Change: In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct tag name never hides the name of an object or function in an outer scope.

Example:
```c
int x[99];
void f() {
    struct x { int a; };
    sizeof(x); // size of the array in C *
    /* size of the struct in C++ */
}
```

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: 11.4.4.2

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: 11.4.9

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: 11.4.10

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:

```c
struct Y;
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.

---

**5** **Affected subclause:** 11.4.11

**Change:** In C++, a typedef name may not be redeclared in a class definition after being used in that definition.

Example:

```c
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.5.7 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. Each implementation must choose the behavior that will be most useful to its marketplace.

**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**

**Affected subclause:** 15.11

This subclause 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.6.1 Modifications to headers

For compatibility with the C standard library, the C++ standard library provides the C headers enumerated in D.9, 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.6.2 Modifications to definitions

C.6.2.1 Types `char16_t` and `char32_t`

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 this International Standard (5.11). They do not appear as macro or type names defined in `<cuchar>` (21.5.5).

C.6.2.2 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 this International Standard (5.11). It does not appear as a macro or type name defined in any of `<cassert>` (19.3.1), `<cstddef>` (17.2.1), `<cstdlib>` (17.2.2), or `<cwchar>` (21.5.4).

C.6.2.3 Header `<assert.h>`

The token `static_assert` is a keyword in this International Standard (5.11). It does not appear as a macro name defined in `<cassert>` (19.3.1).

C.6.2.4 Header `<iso646.h>`

The tokens `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not`, `not_eq`, `or`, `or_eq`, `xor`, and `xor_eq` are keywords in this International Standard (5.11), and are not introduced as macros by `<iso646.h>` (D.9.2).

C.6.2.5 Header `<stdalign.h>`

The token `alignas` is a keyword in this International Standard (5.11), and is not introduced as a macro by `<stdalign.h>` (D.9.3).

C.6.2.6 Header `<stdbool.h>`

The tokens `bool`, `true`, and `false` are keywords in this International Standard (5.11), and are not introduced as macros by `<stdbool.h>` (D.9.4).

C.6.2.7 Macro `NULL`

The macro `NULL`, defined in any of `<clocale>` (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 C++ null pointer constant in this International Standard (17.2).

C.6.3 Modifications to declarations

C.6.3.1 Header `<cstring>`

The following functions have different declarations:

- `strchr`
- `strpbrk`
- `strrchr`
- `strstr`
- `memchr`

Subclause 21.5.3 describes the changes.

C.6.3.2 Header `<cwchar>`

The following functions have different declarations:

- `wcschr`
- `wcpbrk`
- `wcsrchr`
- `wcsstr`
Subclause 21.5.4 describes the changes.

3 Header `<cstddef>` (17.2.1) declares the name `nullptr_t` in addition to the names declared in `<stddef.h>` (D.9) in the C standard library.

C.6.4 Modifications to behavior [diff.mods.to.behavior]

1 Header `<cstdlib>` (17.2.2): The following functions have different behavior:

   (1.1) — `atexit`
   (1.2) — `exit`
   (1.3) — `abort`

Subclause 17.5 describes the changes.

2 Header `<csetjmp>` (17.13.2): The following functions have different behavior:

   (2.1) — `longjmp`

Subclause 17.13.2 describes the changes.

C.6.4.1 Macro `offsetof(type, member-designator)` [diff.offsetof]

1 The macro `offsetof`, defined in `<cstddef>` (17.2.1), accepts a restricted set of `type` arguments in this International Standard. Subclause 17.2.4 describes the change.

C.6.4.2 Memory allocation functions [diff.malloc]

1 The functions `aligned_alloc`, `calloc`, `malloc`, and `realloc` are restricted in this International Standard. Subclause 20.10.12 describes the changes.
Annex D  (normative)
Compatibility features  [depr]

1 This Clause 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 edition of this International Standard, 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.1 Arithmetic conversion on enumerations  [depr.arith.conv.enum]

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. [Note: Three-way comparisons (7.6.8) between such operands are ill-formed. — end note] [Example:]

```cpp
enum E1 { e };  
enum E2 { f };  
bool b = e <= 3.7;  // deprecated  
int k = f - e;      // deprecated  
auto cmp = e <=> f; // error
```

D.2 Implicit capture of *this by reference  [depr.capture.this]

For compatibility with prior C++ International Standards, a lambda-expression with capture-default = (7.5.5.2) may implicitly capture *this by reference. [Example:]

```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.3 Comma operator in subscript expressions  [depr.comma.subscript]

A comma expression (7.6.20) appearing as the expr-or-braced-init-list of a subscripting expression (7.6.1.1) is deprecated. [Note: A parenthesized comma expression is not deprecated. — end note] [Example:]

```cpp
void f(int *a, int b, int c) {
  a[b,c];  // deprecated
  a[(b,c)]; // OK
}
```

D.4 Array comparisons  [depr.array.comp]

Equality and relational comparisons (7.6.10, 7.6.9) between two operands of array type are deprecated. [Note: Three-way comparisons (7.6.8) between such operands are ill-formed. — end note] [Example:]

```cpp
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
```

§ D.4
D.5 Deprecated volatile types

1 Postfix ++ and -- expressions (7.6.1.5) and prefix ++ and -- expressions (7.6.2.2) of volatile-qualified arithmetic and pointer types are deprecated.

[Example:
  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:
  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 (9.3.3.5) with a parameter with volatile-qualified type or with a volatile-qualified return type is deprecated.

[Example:
  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:
  struct linhenykus { short forelimb; };
  void park(linhenykus alvarezsauroid) {
    volatile auto [what_is_this] = alvarezsauroid;  // deprecated
    // ...
  }
  — end example]

D.6 Redeclaration of static constexpr data members

1 For compatibility with prior C++ International Standards, a constexpr static data member may be redundantly redeclared outside the class with no initializer. This usage is deprecated. [Example:
  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.7 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: Such a declaration in an importable module unit is ill-formed. — end note] [Example:
  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.8 Implicit declaration of copy functions

The implicit definition of a copy constructor (11.4.4.2) as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor (11.4.6). The implicit definition of a copy assignment operator (11.4.5) as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor. In a future revision of this International Standard, these implicit definitions could become deleted (9.5.3).

D.9 C headers

For compatibility with the C standard library, the C++ standard library provides the C headers shown in Table 147.

Table 147: C headers

| <assert.h> | <inttypes.h> | <signal.h> | <stdio.h> | <wchar.h> |
| <complex.h> | <iso646.h> | <stdalign.h> | <stdlib.h> | <wctype.h> |
| <ctype.h> | <limits.h> | <stdarg.h> | <stdbool.h> | <stddef.h> |
| <errno.h> | <locale.h> | <stdlib.h> | <string.h> | <uchar.h> |
| <fenv.h> | <math.h> | <setjmp.h> | <stdio.h> | <tgmath.h> |
| <float.h> | <string.h> | <time.h> | <uchar.h> |

D.9.1 Header <complex.h> synopsis

The header <complex.h> behaves as if it simply includes the header <complex> (26.4.1).

D.9.2 Header <iso646.h> synopsis

The C++ header <iso646.h> is empty. [Note: and, and_eq, bitand, bitor, compl, not_eq, not, or, or_eq, xor, and xor_eq are keywords in this International Standard (5.11). —end note]

D.9.3 Header <stdalign.h> synopsis

The contents of the C++ header <stdalign.h> are the same as the C standard library header <stdalign.h>, with the following changes: The header <stdalign.h> does not define a macro named alignas.

See also: ISO C 7.15

D.9.4 Header <stdbool.h> synopsis

The contents of the C++ header <stdbool.h> are the same as the C standard library header <stdbool.h>, with the following changes: The header <stdbool.h> does not define macros named bool, true, or false.

See also: ISO C 7.18

D.9.5 Header <tgmath.h> synopsis

The header <tgmath.h> behaves as if it simply includes the headers <cmath> (26.8.1) and <complex> (26.4.1).

[Note: The overloads provided in C by type-generic macros are already provided in <complex> and <cmath> by “sufficient” additional overloads. —end note]
D.9.6 Other C headers

1 Every C header other than `<complex.h>` (D.9.1), `<iso646.h>` (D.9.2), `<stdalign.h>` (D.9.3), `<stdbool.h>` (D.9.4), and `<tgmath.h>` (D.9.5), 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 `<c_name>` 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: The header `<stdlib>` 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.10 Requires paragraph

1 In addition to the elements specified in 16.4.1.4, 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.11 Relational operators

1 The header `<utility>` (20.2.1) has the following additions:

   ```cpp
   namespace std::rel_ops {
   template<class T> bool operator!=(const T&, const T&);  
   template<class T> bool operator> (const T&, const T&);    
   template<class T> bool operator<=(const T&, const T&);    
   template<class T> bool operator>=(const T&, const T&);    
   }
   ```

2 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

   ```cpp
   template<class T> bool operator!=(const T& x, const T& y);
   ```

   3 `Requires:` Type `T` is `Cpp17EqualityComparable` (Table 25).

   4 `Returns:` !(x == y).

   ```cpp
   template<class T> bool operator>(const T& x, const T& y);
   ```

   5 `Requires:` Type `T` is `Cpp17LessThanComparable` (Table 26).

   6 `Returns:` y < x.

   ```cpp
   template<class T> bool operator<=(const T& x, const T& y);
   ```

   7 `Requires:` Type `T` is `Cpp17LessThanComparable` (Table 26).

   8 `Returns:` !(y < x).

   ```cpp
   template<class T> bool operator>=(const T& x, const T& y);
   ```

   9 `Requires:` Type `T` is `Cpp17LessThanComparable` (Table 26).

   10 `Returns:` !(x < y).

D.12 char* streams

D.12.1 Header `<strstream>` synopsis

1 The header `<strstream>` defines types that associate stream buffers with character array objects and assist reading and writing such objects.

   ```cpp
   namespace std {
   class strstreambuf;
   class istrstream;
   class ostrstream;
   }
class strstream;
}

D.12.2 Class strstreambuf

namespace std {

class strstreambuf : public basic_streambuf<char> {
public:
   strstreambuf() : strstreambuf(0) {}
   explicit strstreambuf(streamsize alsize_arg);
   strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
   strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = nullptr);
   strstreambuf(const char* gnext_arg, streamsize n);
   strstreambuf(signed char* gnext_arg, streamsize n,
                signed char* pbeg_arg = nullptr);
   strstreambuf(const signed char* gnext_arg, streamsize n);
   strstreambuf(unsigned char* gnext_arg, streamsize n,
                unsigned char* pbeg_arg = nullptr);
   strstreambuf(const unsigned char* gnext_arg, streamsize n);
virtual ~strstreambuf();

   void freeze(bool freezefl = true);
   char* str();
   int pcount();

protected:
   int_type overflow (int_type c = EOF) override;
   int_type pbackfail(int_type c = EOF) override;
   int_type underflow() 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;
   streambuf* setbuf(char* s, streamsize n) override;

private:
   using strstate = T1; // exposition only
   static const strstate allocated; // exposition only
   static const strstate constant; // exposition only
   static const strstate dynamic; // exposition only
   static const strstate frozen; // exposition only
   strstate strmode; // exposition only
   streamsize alsize; // exposition only
   void* (*palloc)(size_t); // exposition only
   void (*pfree)(void*); // exposition only
};

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.

[Note: For the sake of exposition, these are represented as elements of a bitmask type (indicated here as T1) called strstate. The elements are:

(2.1) — allocated, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the strstreambuf object;
(2.2) — constant, set when the array object has const elements, so the output sequence cannot be written;
(2.3) — dynamic, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;
(2.4) — frozen, set when the program has requested that the array object not be altered, reallocated, or freed.
— end note]
Note: For the sake of exposition, the maintained data is presented here as:

- \( \text{strstate strmode} \), the attributes of the array object associated with the \text{strstreambuf} object;
- \( \text{int alsiz} \), the suggested minimum size for a dynamic array object;
- \( \text{void* (}*\text{palloc}(\text{size_t}) \), points to the function to call to allocate a dynamic array object;
- \( \text{void (}*\text{pfree}(\text{void*}) \), points to the function to call to free a dynamic array object.

Each object of class \text{strstreambuf} has a \textit{seekable area}, delimited by the pointers \text{seeklow} and \text{seekhigh}. If \text{gnext} is a null pointer, the seekable area is undefined. Otherwise, \text{seeklow} equals \text{gbeg} and \text{seekhigh} is either \text{pend}, if \text{pend} is not a null pointer, or \text{gend}.

D.12.2.1 \text{strstreambuf} constructors

\[\text{explicit strstreambuf(streamsize alsiz}_\text{arg});\]

Effects: Initializes the base class with \text{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>\text{strmode}</td>
<td>dynamic</td>
</tr>
<tr>
<td>\text{alsiz}</td>
<td>\text{alsiz}_\text{arg}</td>
</tr>
<tr>
<td>\text{palloc}</td>
<td>a null pointer</td>
</tr>
<tr>
<td>\text{pfree}</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

\[\text{strstreambuf( void* (}*\text{palloc}_\text{arg}(\text{size_t}) \), \text{void (}*\text{pfree}_\text{arg}(\text{void*}) );\]

Effects: Initializes the base class with \text{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>\text{strmode}</td>
<td>dynamic</td>
</tr>
<tr>
<td>\text{alsiz}</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>\text{palloc}</td>
<td>\text{palloc}_\text{arg}</td>
</tr>
<tr>
<td>\text{pfree}</td>
<td>\text{pfree}_\text{arg}</td>
</tr>
</tbody>
</table>

\[\text{strstreambuf(char* gnext}_\text{arg}, \text{streamsize n}, \text{char* pbeg}_\text{arg} = \text{nullptr});\]
\[\text{strstreambuf(signed char* gnext}_\text{arg}, \text{streamsize n}, \text{signed char* pbeg}_\text{arg} = \text{nullptr});\]
\[\text{strstreambuf(unsigned char* gnext}_\text{arg}, \text{streamsize n}, \text{unsigned char* pbeg}_\text{arg} = \text{nullptr});\]

Effects: Initializes the base class with \text{streambuf}(\). The postconditions of this function are indicated in Table 150.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{strmode}</td>
<td>0</td>
</tr>
<tr>
<td>\text{alsiz}</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>\text{palloc}</td>
<td>a null pointer</td>
</tr>
<tr>
<td>\text{pfree}</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>
`gnext_arg` shall point to the first element of an array object whose number of elements $N$ is determined as follows:

1. If $n > 0$, $N$ is $n$.
2. If $n == 0$, $N$ is `std::strlen(gnext_arg)`.
3. If $n < 0$, $N$ is `INT_MAX`.

If `pbeg_arg` is a null pointer, the function executes:

```c
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

Otherwise, the function executes:

```c
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
```

`strstreambuf(const char* gnext_arg, streamsize n);`  
`strstreambuf(const signed char* gnext_arg, streamsize n);`  
`strstreambuf(const unsigned char* gnext_arg, streamsize n);`

Effects: Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets constant in `strmode`.

virtual ~strstreambuf();

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.12.2.3 describes how a dynamically allocated array object is freed.)

D.12.2.2 Member functions  

```c
void freeze(bool freezefl = true);
```

Effects: If `strmode & dynamic` is nonzero, alters the freeze status of the dynamic array object as follows:

1. If `freezefl` is `true`, the function sets `frozen` in `strmode`.
2. Otherwise, it clears `frozen` in `strmode`.

```c
char* str();
```

Effects: Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

Remarks: The return value can be a null pointer.

```c
int pcount() const;
```

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.12.2.3 `strstreambuf` overridden virtual functions  

```c
int_type overflow(int_type c = EOF) override;
```

Effects: Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

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`.
2. If `c == EOF`, there is no character to append. Returns a value other than `EOF`.

Remarks: The function can alter the number of write positions available as a result of any call.

---

327 The function signature `strlen(const char*)` is declared in `<cstring>` (21.5.3). The macro `INT_MAX` is defined in `<climits>` (17.3.6).
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.\(^{328}\) If \( palloc \) is not a null pointer, the function calls \((\ast palloc)(n)\) 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 \( pfree \) is not a null pointer, the function calls \((\ast pfree)(p)\). 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.

\[
\text{int_type pbackfail(int_type c = EOF) override;}
\]

Puts back the character designated by \( c \) to the input sequence, if possible, in one of three ways:

\begin{enumerate}[\( (7.1) \)]
\item If \( c != \text{EOF} \), if the input sequence has a putback position available, and if \((\ast\text{char})c == \ast\text{gnext}[-1] \), assigns \( \ast\text{gnext} - 1 \) to \( \text{gnext} \). Returns \( c \).
\item If \( c != \text{EOF} \), if the input sequence has a putback position available, and if \( \text{strmode} & \text{constant} \) is zero, assigns \( c \) to \( \ast\ast\text{gnext} \). Returns \( c \).
\item If \( c == \text{EOF} \) and if the input sequence has a putback position available, assigns \( \text{gnext} - 1 \) to \( \text{gnext} \). Returns a value other than EOF.
\end{enumerate}

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:

\begin{enumerate}[\( (10.1) \)]
\item If the input sequence has a read position available, the function signals success by returning \( (\ast\text{unsigned char})\ast\text{gnext} \).
\item 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 \( (\ast\text{unsigned char})\ast\text{gnext} \).
\end{enumerate}

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 \( \text{xnext} \).

Returns: \( \text{pos_type}(\text{newoff}) \), constructed from the resultant offset \( \text{newoff} \) (of type \( \text{off_type} \)), 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 \( \text{pos_type}(\text{off_type}(-1)) \).

\(^{328}\) An implementation should consider \( \text{alsize} \) in making this decision.
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>way == ios::beg or way == ios::end</td>
<td></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>xnext - xbeg</td>
</tr>
<tr>
<td>way == ios::end</td>
<td>seekhigh - xbeg</td>
</tr>
</tbody>
</table>

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).

(17.1) If (which & ios::in) != 0, positions the input sequence.
(17.2) If (which & ios::out) != 0, positions the output sequence.
(17.3) 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():

(18.1) If newoff is an invalid stream position, has a negative value, or has a value greater than (seekhigh - seeklow), the positioning operation fails.
(18.2) 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.12.3 Class istrstream

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();

            strstreambuf* rdbuf() const;
            char* str();
        private:
            strstreambuf sb;           // exposition only
    };
};
The class `istringstream` 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. (1.1) — `sb`, the `strstreambuf` object.

### D.12.3.1 `istringstream` constructors

```c
explicit istringstream(const char* s);
explicit istringstream(char* s);
```

*Effects:* Initializes the base class with `istream(&sb)` and `sb` with `strstreambuf(s, 0)`. `s` shall designate the first element of an NTBS.

```c
istringstream(const char* s, streamsize n);
istringstream(char* s, streamsize n);
```

*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.12.3.2 Member functions

```c
strstreambuf* rdbuf() const;
```

*Returns:* `const_cast<strstreambuf*>(&sb)`.

```c
char* str();
```

*Returns:* `rdbuf()->str()`.

### D.12.4 Class `ostrstream`

```c
namespace std {
    class ostrstream : public basic_ostream<char> {
        public:
            ostrstream();
            ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);  
            virtual ~ostrstream();
            strstreambuf* rdbuf() const;
            void freeze(bool freeze1 = true);
            char* str();
            int pcount() const;
        private:
            strstreambuf sb; // exposition only
    };
}
```

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. (1.1) — `sb`, the `strstreambuf` object.

### D.12.4.1 `ostrstream` constructors

```c
ostrstream();
```

*Effects:* Initializes the base class with `ostream(&sb)` and `sb` with `strstreambuf()`.

```c
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:

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))`.\(^{329}\)

\(^{329}\) The function signature `strlen(const char*)` is declared in `<cstring>` (21.5.3).
D.12.4.2 Member functions

strstreambuf* rdbuf() const;

1 Returns: (strstreambuf*)&sb.

void freeze(bool freezefl = true);

2 Effects: Calls rdbuf()->freeze(freezefl).

char* str();

3 Returns: rdbuf()->str().

int pcount() const;

4 Returns: rdbuf()->pcount().

D.12.5 Class strstream

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.

D.12.5.1 strstream constructors

strstream();

1 Effects: Initializes the base class with iostream(&sb).

strstream(char* s, int n,
    ios_base::openmode mode = ios_base::in|ios_base::out);

2 Effects: Initializes the base class with iostream(&sb), and sb with one of the 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.12.5.2 strstream destructor

virtual ~strstream();

Effects: Destroys an object of class strstream.

D.12.5.3 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.13 Deprecated type traits

The header <type_traits> (20.15.2) has the following addition:

namespace std {
  template<class T> struct is_pod;
  template<class T> inline constexpr bool is_pod_v = is_pod<T>::value;
}

The behavior of a program that adds specializations for any of the templates defined in this subclause is undefined, unless explicitly permitted by the specification of the corresponding template.

template<class T> struct is_pod;

Requires: remove_all_extents_t<T> shall be a complete type or cv void.

is_pod<T> is a Cpp17UnaryTypeTrait (20.15.1) 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: It is unspecified whether a closure type (7.5.5.1) is a POD type. — end note]

D.14 Tuple

The header <tuple> (20.5.2) has the following additions:

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:

(4.1) — for the first specialization, add_volatile_t<TE>, and

(4.2) — 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.15 Variant [depr.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>;
}

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.16 Deprecated iterator primitives [depr.iterator.primitives]

D.16.1 Basic iterator [depr.iterator.basic]

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: 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: 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:
class MyIterator :
  public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
    // code implementing ++, etc.
  };

--- end example

### D.17 Deprecated move_iterator access

The following member is declared in addition to those members specified in 23.5.3.5:

```
namespace std {
  template<class Iterator>
  class move_iterator {
    public:
      constexpr pointer operator->() const;
    };
}
```

1) The following member is declared in addition to those members specified in 23.5.3.5:

```
constexpr pointer operator->() const;
```

2) Returns: current.

### D.18 Deprecated shared_ptr atomic access

The header `<memory>` (20.10.2) has the following additions:

```
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);
  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);
}
```

2) 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.

3) The meaning of the arguments of type memory_order is explained in 31.4.

```
template<class T> bool atomic_is_lock_free(const shared_ptr<T>* p);
```

4) Requires: `p` shall not be null.

5) Returns: true if atomic access to `∗p` is lock-free, false otherwise.

§ D.18
template<class T> shared_ptr<T> atomic_load(const shared_ptr<T>* p);
   Requires: p shall not be null.
   Returns: atomic_load_explicit(p, memory_order::seq_cst).
   Throws: Nothing.

template<class T> shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
   Requires: p shall not be null.
   Requires: mo shall not be memory_order::release or memory_order::acq_rel.
   Returns: *p.
   Throws: Nothing.

template<class T> void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
   Requires: p shall not be null.
   Effects: As if by atomic_store_explicit(p, r, memory_order::seq_cst).
   Throws: Nothing.

template<class T> void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
   Requires: p shall not be null.
   Requires: mo shall not be memory_order::acquire or memory_order::acq_rel.
   Effects: As if by p->swap(r).
   Throws: Nothing.

template<class T> shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
   Requires: p shall not be null.
   Returns: atomic_exchange_explicit(p, r, memory_order::seq_cst).
   Throws: Nothing.

shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
   Requires: p shall not be null.
   Effects: As if by p->swap(r).
   Returns: The previous value of *p.
   Throws: Nothing.

template<class T>
bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
   Requires: p shall not be null and v shall not be null.
   Returns: atomic_compare_exchange_weak_explicit(p, v, w, memory_order::seq_cst, memory_order::seq_cst)
   Throws: Nothing.

template<class T>
bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
   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);

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.

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.

Returns: true if *p was equivalent to *v, false otherwise.

Throws: Nothing.

Remarks: Two shared_ptr objects are equivalent if they store the same pointer value and share ownership. The weak form may fail spuriously. See 31.8.1.

D.19 Deprecated basic_string capacity

The following member is declared in addition to those members specified in 21.3.2.4:

namespace std {
    template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
    class basic_string {
        public:
            void reserve();
    };
}

void reserve();

Effects: After this call, capacity() has an unspecified value greater than or equal to size(). [Note: This is a non-binding shrink to fit request. —end note]

D.20 Deprecated standard code conversion facets

The header <codecvt> provides code conversion facets for various character encodings.

D.20.1 Header <codecvt> synopsis

namespace std {
    enum codecvt_mode {
        consume_header = 4,
        generate_header = 2,
        little_endian = 1
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8 : public codecvt<Elem, char, mbstate_t> {
        public:
            explicit codecvt_utf8(size_t refs = 0);
            ~codecvt_utf8();
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf16 : public codecvt<Elem, char, mbstate_t> {
        public:
            explicit codecvt_utf16(size_t refs = 0);
            ~codecvt_utf16();
    };

    template<class Elem, unsigned long Maxcode = 0x10ffff, codecvt_mode Mode = (codecvt_mode)0>
    class codecvt_utf8_utf16 : public codecvt<Elem, char, mbstate_t> {
        public:
            explicit codecvt_utf8_utf16(size_t refs = 0);
            ~codecvt_utf8_utf16();
    };

§ D.20.1 1676
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.

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.

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.

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 10646-1:1993.

### D.21 Deprecated convenience conversion interfaces

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.21.1 Class template wstring_convert

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:* 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

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(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:

- **byte_err_string** — a byte string to display on errors
- **wide_err_string** — a wide string to display on errors
- **cvtptr** — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
using byte_string = basic_string<char, char_traits<char>, ByteAlloc>;

The type shall be a synonym for basic_string<char, char_traits<char>, ByteAlloc>.

size_t converted() const noexcept;

Returns: cvtcount.

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:

(7.1) — 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.

(7.2) — 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.

using state_type = typename Codecvt::state_type;

The type shall be a synonym for Codecvt::state_type.

state_type state() const;

returns cvtstate.

using state_type = typename Codecvt::state_type;

The type shall be a synonym for Codecvt::state_type.

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:

(13.1) — 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.

(13.2) — 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 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.21.2 Class template wbuffer_convert [ depr.conversions.buffer]

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();
            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.22 Deprecated locale category facets
[depr.locale.category]

The ctype locale category includes the following facets as if they were specified in table Table 102 of 28.3.1.1.1.
    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 28.3.1.1.1.
    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 28.4.1.4. 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.23 Deprecated filesystem path factory functions
[depr.fs.path.factory]

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 29.11.7.3.

    Returns:

(2.1) — If value_type is char and the current native narrow encoding (29.11.7.2.2) 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 (29.11.7.2.1) applies to the arguments for these functions. How Unicode encoding conversions are performed is unspecified.

[Example: 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

—end note

D.24 Deprecated atomic operations

1 The header `<atomics>` (31.2) has the following additions.

```cpp
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.24.1 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:

```cpp
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* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) volatile noexcept;
```

D.24.2 Non-member functions

```cpp
template<class T>
void atomic_init(volatile atomic<T>*, object, typename atomic<T>::value_type desired) noexcept;
template<class T>
void atomic_init(atomic<T>*, object, typename atomic<T>::value_type desired) noexcept;
```

1 Effects: Equivalent to: `atomic_store_explicit(object, desired, memory_order::relaxed);`

D.24.3 Operations on atomic types

```cpp
#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: This operation may need to initialize locks. — end note] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [Example:

```cpp
atomic<int> v = ATOMIC_VAR_INIT(5);
```
D.24.4 Flag type and operations

#define ATOMIC_FLAG_INIT see below

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.
Bibliography

The following documents are cited informatively in this document.

— IANA Time Zone Database, available at https://www.iana.org/time-zones
— ISO 4217:2015, Codes for the representation of currencies

The arithmetic specification described in ISO/IEC 10967-1:2012 is called LIA-1 in this document.
Cross references

This annex lists each clause or subclause label and the corresponding clause or subclause number and page number, in alphabetical order by label.

accumulate (25.10.2) 1125
adjacent.difference (25.10.11) 1132
adjustfield.manip (29.5.6.2) 1374
alg.adjacent.find (25.6.8) 1072
alg.all.of (25.6.1) 1066
alg.any.of (25.6.2) 1067
alg.binary.search (25.8.3) 1100
alg.c.library (25.12) 1140
alg.clamp (25.8.9) 1118
alg.copy (25.7.1) 1078
alg.count (25.6.9) 1072
alg.equal (25.6.11) 1074
alg.fill (25.7.6) 1086
alg.find (25.6.5) 1069
alg.find.end (25.6.6) 1070
alg.find.first.of (25.6.7) 1071
alg.foreach (25.6.4) 1068
alg.generate (25.7.7) 1086
alg.heap.operations (25.8.7) 1112
alg.is.permutation (25.6.12) 1076
alg.lex.comparison (25.8.10) 1119
alg.merge (25.8.5) 1105
alg.min.max (25.8.8) 1115
alg.modifying.operations (25.7) 1078
alg.move (25.7.2) 1080
alg.none.of (25.6.3) 1067
alg.nonmodifying (25.6) 1066
alg.nth.element (25.8.2) 1099
alg.partitions (25.8.4) 1102
alg.permutation.generators (25.8.12) 1120
alg.random.sample (25.7.12) 1093
alg.random.shuffle (25.7.13) 1093
alg.remove (25.7.8) 1087
alg.replace (25.7.5) 1084
alg.req (23.3.7) 926
alg.req.general (23.3.7.1) 926
alg.req.ind.cmp (23.3.7.5) 927
alg.req.ind.copy (23.3.7.3) 927
alg.req.ind.move (23.3.7.2) 926
alg.req.ind.swap (23.3.7.4) 927
alg.req.mergeable (23.3.7.7) 928
alg.req.permutable (23.3.7.6) 927
alg.req.sortable (23.3.7.8) 928
alg.reverse (25.7.10) 1091
alg.rotate (25.7.11) 1092
alg.search (25.6.13) 1077
alg.set.operations (25.8.6) 1107
alg.shift (25.7.14) 1094
alg.sort (25.8.1) 1095
alg.sorting (25.8) 1094
alg.swap (25.7.3) 1082
alg.three.way (25.8.11) 1120
alg.transform (25.7.4) 1082
alg.unique (25.7.9) 1089
algorithm.stable (16.5.5.8) 486
algorithm.syn (25.4) 1029
algorithm: (Clause 25) 1024
algorithms: (Clause 25) 1024
algorithms: (Clause 25) 1026
algorithms: parallel (25.3) 1026
algorithms: parallel: defns (25.3.1) 1026
algorithms: parallel: exceptions (25.3.4) 1028
algorithms: parallel: exec (25.3.3) 1027
algorithms: parallel: overloads (25.3.5) 1028
algorithms: parallel: user (25.3.2) 1027
algorithms: requirements (25.2) 1024
algorithms: results (25.5) 1064
alloc.errors (17.6.3) 509
allocator.adaptor (20.13) 662
allocator.adaptor.cnstr (20.13.3) 664
allocator.adaptor.members (20.13.4) 665
allocator.adaptor.syn (20.13.1) 662
allocator.adaptor.types (20.13.2) 664
allocator.globals (20.10.10.2) 630
allocator.members (20.10.10.1) 629
allocator.requirements (16.5.3.5) 475
allocator.requirements: completeness (16.5.3.5.1) 479
allocator.tag (20.10.7) 624
allocator: traits (20.10.9) 627
allocator: traits: members (20.10.9.2) 628
allocator: traits: types (20.10.9.1) 628
allocator: uses (20.10.8) 625
allocator: uses: construction (20.10.8.2) 625
allocator: uses: trait (20.10.8.1) 625
alt.headers (16.5.4.1) 482
any (20.8) 604
any: assign (20.8.3.2) 606
any: bad: any: cast (20.8.2) 604
any: class (20.8.3) 605
any: cons (20.8.3.1) 605
any: modifiers (20.8.3.3) 607
any: nonmembers (20.8.4) 608
any: observers (20.8.3.4) 607
any: synop (20.8.1) 604
arithmetic.operations (20.14.6) 671
arithmetic.operations: divides (20.14.6.4) 672
arithmetic.operations: minius (20.14.6.2) 671
arithmetic.operations: modulus (20.14.6.5) 672
arithmetic.operations: multiplies (20.14.6.3) 672
arithmetic.operations: negate (20.14.6.6) 672

Cross references 1685
locale.ctype.virtuals (28.4.1.1.2) 1324
locale.facet (28.3.1.1.2) 1319
locale.global.templates (28.3.2) 1322
locale.id (28.3.1.1.3) 1320
locale.messages (28.4.7.1) 1352
locale.messages.byname (28.4.7.2) 1353
locale.messages.members (28.4.7.1.1) 1352
locale.messages.virtuals (28.4.7.1.2) 1353
locale.money.get (28.4.6.1) 1347
locale.money.get.members (28.4.6.1.1) 1347
locale.money.get.virtuals (28.4.6.1.2) 1347
locale.money.put (28.4.6.2) 1348
locale.money.put.members (28.4.6.2.1) 1349
locale.money.put.virtuals (28.4.6.2.2) 1349
locale.moneypunct (28.4.6.3) 1349
locale.moneypunct.byname (28.4.6.4) 1352
locale.moneypunct.members (28.4.6.3.1) 1351
locale.moneypunct.virtuals (28.4.6.3.2) 1351
locale.nm.put (28.4.2.2) 1335
locale.num.get (28.4.2.1) 1332
locale.numpunct (28.4.3.1) 1338
locale.numpunct.byname (28.4.3.2) 1340
locale.numpunct.members (28.4.3.1.2) 1343
locale.numpunct.virtuals (28.4.3.1.3) 1343
locale.time.get (28.4.5.1) 1342
locale.time.get.members (28.4.5.1.1) 1343
locale.time.get.virtuals (28.4.5.1.2) 1344
locale.time.put (28.4.5.3) 1345
locale.time.put.members (28.4.5.3.1) 1346
locale.time.put.virtuals (28.4.5.3.2) 1346
locale.types (28.3.1.1) 1318
locales (28.3) 1316
localization (Clause 28) 1315
localization.general (28.1) 1315
logic.error (19.2.2) 547
logical.operations (20.14.9) 677
logical.operations.and (20.14.9.1) 677
logical.operations.not (20.14.9.3) 677
logical.operations.or (20.14.9.2) 677
lower.bound (25.8.3.1) 1100

macro.names (16.5.4.3.2) 482
make.heap (25.8.7.3) 1113
map (22.4.4) 849
map.access (22.4.4.3) 852
map.cons (22.4.4.2) 852
map.erase (22.4.4.5) 854
map.modifiers (22.4.4.4) 852
map.overview (22.4.4.1) 849
mask.array.assign (26.7.8.2) 1207
mask.array.comp.assign (26.7.8.3) 1207
mask.array.fill (26.7.8.4) 1207
math.constants (26.9.2) 1224
mem.poly.alloc.class (20.12.3) 655
mem.poly.allocctor (20.12.3) 656
mem.poly.alloc.class (20.12.3.3) 657
mem.poly.allocctor (20.12.3.2) 656
mem.res (20.12) 653
mem.res.class (20.12.2) 654
mem.res.eq (20.12.2.3) 655
mem.res.global (20.12.4) 657
mem.res.monotonic.buffer (20.12.6) 660
mem.res.monotonic.buffer.ctor (20.12.6.1) 661
mem.res.monotonic.buffer.mem (20.12.6.2) 662
mem.res.pool (20.12.5) 658
mem.res.pool.ctor (20.12.5.3) 659
mem.res.pool.mem (20.12.5.4) 660
mem.res.pool.options (20.12.5.2) 659
mem.res.pool.overview (20.12.5.1) 658
mem.res.private (20.12.2.2) 654
mem.res.public (20.12.2.1) 654
mem.res.syn (20.12.1) 653
member.functions (16.5.5.5) 485
memory (20.10) 614
memory.general (20.10.1) 614
memory.syn (20.10.2) 615
meta (20.15) 689
meta.const.eval (20.15.10) 713
meta.help (20.15.1) 696
meta.logical (20.15.8) 711
meta.member (20.15.9) 712
meta.rel (20.15.6) 704
meta.rqmts (20.15.1) 689
meta.trans (20.15.7) 706
meta.trans.arr (20.15.7.4) 707
meta.trans.cv (20.15.7.1) 706
meta.trans.other (20.15.7.6) 708
meta.trans.ptr (20.15.7.5) 708
meta.trans.ref (20.15.7.2) 706
meta.trans.sign (20.15.7.3) 707
meta.type.synop (20.15.2) 689
meta.unary (20.15.4) 696
meta.unary.cat (20.15.4.1) 696
meta.unary.comp (20.15.4.2) 697
meta.unary.prop (20.15.4.3) 697
meta.unary.prop.query (20.15.5) 703
mismatch (25.6.10) 1073
module (Clause 10) 236
module.context (10.6) 244
module.global.frag (10.4) 241
module.import (10.3) 240
module.interface (10.2) 237
module.private.frag (10.5) 243
module.reach (10.7) 245
module.unit (10.1) 236
move.iter.cons (23.5.3.3) 940
move.iter.elem (23.5.3.5) 940
move.iter.nav (23.5.3.6) 940
move.iter.nonmember (23.5.3.8) 940
move.iter.op.comp (23.5.3.7) 941
move.iter.op.conv (23.5.3.4) 940
move.iter.requirements (23.5.3.2) 940
move.iterator (23.5.3.1) 939
move.iterators (23.5.3) 938

Cross references 1696
move.sent.ops (23.5.3.10) 943
move.sentinel (23.5.3.9) 942
multibyte.strings (16.4.2.2.5.2) 467
multimap (22.4.5) 854
multimap.cons (22.4.5.2) 857
multimap.erase (22.4.5.4) 857
multimap.modifiers (22.4.5.3) 857
multimap.overview (22.4.5.1) 854
multiset (22.4.7) 861
multiset.cons (22.4.7.2) 864
multiset.erase (22.4.7.3) 864
multiset.overview (22.4.7.1) 861
mutex.syn (32.5.1) 1565
namespace.alias (9.8.2) 218
namespace.constraints (16.5.4.2) 480
namespace.def (9.8.1) 214
namespace.future (16.5.4.2.3) 480
namespace.memdef (9.8.1.2) 216
namespaceposix (16.5.4.2.2) 480
namespace.qual (6.5.3.2) 46
namespace.std (16.5.4.2.1) 480
namespace.udecl (9.9) 220
namespace.udir (9.8.3) 218
namespace.unnamed (9.8.1.1) 216
narrow.stream.objects (29.4.3) 1359
new.badlength (17.6.3.2) 510
new.delete (17.6.2) 505
new.delete.array (17.6.2.2) 507
new.delete.dataraces (17.6.2.4) 509
new.delete.placement (17.6.2.3) 509
new.delete.single (17.6.2.1) 505
new.handler (17.6.3.3) 510
new.syn (17.6.1) 504
nullable.pointer.requirements (16.5.3.3) 474
numarray (26.7) 1190
numbers (26.9) 1224
numbers.syn (26.9.1) 1224
numeric.iota (25.10.12) 1133
numeric.limits (17.3.5) 495
numeric.limits.members (17.3.5.1) 496
numeric.ops (25.10) 1124
numeric.ops.gcd (25.10.13) 1133
numeric.ops.lcm (25.10.14) 1133
numeric.ops.midpoint (25.10.15) 1134
numeric.ops.overview (25.9) 1121
numeric.requirements (26.2) 1141
numeric.special (17.3.5.2) 500
numerics (Clause 26) 1141
numerics.defns (25.10.1) 1125
numerics.general (26.1) 1141
objects.within.classes (16.4.2.4) 468
ofstream (29.9.4) 1429
ofstream.assign (29.9.4.2) 1430
ofstream.cons (29.9.4.1) 1429
ofstream.members (29.9.4.3) 1430
optional (20.6) 581
optional.assign (20.6.3.3) 585
optional.bad.access (20.6.5) 590
optional.comp.with.t (20.6.8) 591
optional.ctor (20.6.3.1) 583
optional.dtor (20.6.3.2) 585
optional.general (20.6.1) 581
optional.hash (20.6.10) 593
optional.mod (20.6.3.6) 590
optional.nullops (20.6.7) 591
optional.nullopt (20.6.4) 590
optional.observe (20.6.3.5) 589
optional.optional (20.6.3) 582
optional.relops (20.6.6) 590
optional.special (20.6.9) 592
optional.swap (20.6.3.4) 588
optional.syn (20.6.2) 581
organization (16.5.1) 468
ostream (29.7.5.1) 1395
ostream.assign (29.7.5.1.2) 1397
ostream.cons (29.7.5.1.1) 1397
ostream.formatted (29.7.5.2) 1398
ostream.formatted.reqlnts (29.7.5.2.1) 1398
ostream.inserters (29.7.5.2.3) 1400
ostream.inserters.arithmetic (29.7.5.2.2) 1398
ostream.inserters.character (29.7.5.2.4) 1400
ostream.iterator (23.6.2) 953
ostream.iterator.cons.des (23.6.2.1) 954
ostream.iterator.ops (23.6.2.2) 954
ostream.manip (29.7.5.4) 1402
ostream.rvalue (29.7.5.5) 1402
ostream.seek (29.7.5.1.4) 1398
ostream.sentry (29.7.5.1.3) 1397
ostream.syn (29.7.2) 1383
ostream.unformatted (29.7.5.3) 1401
ostreambuf.iterator (23.6.4.1) 957
ostreambuf.iterator.ops (23.6.4.2) 957
ostreambuf.iterator (23.6.4.3) 956
ostreamstream (29.8.4) 1416
ostreamstream.assign (29.8.4.2) 1418
ostreamstream.cons (29.8.4.1) 1417
ostreamstream.members (29.8.4.3) 1418
out.of.range (19.2.6) 548
output.iterators (23.3.5.3) 922
output.streams (29.7.5) 1394
over (Clause 12) 311
over.ass (12.6.2.1) 339
over.best.ics (12.4.3.1) 328
over.binary (12.6.2) 339
over.built (12.7) 341
over.call (12.6.3) 340
over.call.func (12.4.1.1.1) 316
over.call.object (12.4.1.1.2) 317
over.dcl (12.3) 313
over.ics.ellipse (12.4.3.1.3) 331
over.ics.list (12.4.3.1.5) 332
over.ics.rank (12.4.3.2) 334
over.ics.ref (12.4.3.1.4) 331
over.ics.scs (12.4.3.1.1) 330
over.ics.user (12.4.3.1.2) 331
over.inc (12.6.6) 341

Cross references 1697
string::view.iterators (21.4.2.2) 774
string::view.literals (21.4.7) 779
string::view.modifiers (21.4.2.5) 775
string::view.ops (21.4.2.6) 775
string::view.synop (21.4.1) 770
string::view.template (21.4.2) 771
stringbuf (29.8.2) 1407
stringbuf.assign (29.8.2.2) 1410
stringbuf.cons (29.8.2.1) 1409
stringbuf.members (29.8.2.3) 1410
stringbuf.virtuals (29.8.2.4) 1411
strings (Clause 21) 739
strings.general (21.1) 739
stringstream (29.8.5) 1418
stringstream.assign (29.8.5.2) 1420
stringstream.cons (29.8.5.1) 1419
stringstream.members (29.8.5.3) 1420
structure (16.4.1) 462
structure.elements (16.4.1.1) 462
structure.requirements (16.4.1.3) 463
structure.see.also (16.4.1.5) 464
structure.specifications (16.4.1.4) 463
structure.summary (16.4.1.2) 462
support (Clause 17) 488
support.coroutine (17.12) 527
support.dynamic (17.6) 504
support.exception (17.9) 515
support.general (17.1) 488
support.initlist (17.10) 519
support.initlist.access (17.10.3) 519
support.initlist.cons (17.10.2) 519
support.initlist.range (17.10.4) 519
support.limits (17.3) 492
support.limits.general (17.3.1) 492
support.rtti (17.7) 511
support.runtime (17.13) 532
support.signal (17.13.4) 533
support.sysloc (17.8) 513
support.sysloc.class (17.8.2) 513
support.sysloc.cons (17.8.2.1) 514
support.sysloc.obs (17.8.2.2) 515
support.start.term (17.5) 503
support.types (17.2) 488
support.types.byteops (17.2.5) 491
support.types.layout (17.2.4) 490
support.types.nullptr (17.2.3) 490
swappable.requirements (16.5.3.2) 471
syncstream (29.10) 1433
syncstream::syncread (29.10.3) 1435
syncstream::syncbuf (29.10.2) 1433
syncstream::syncbuf.assign (29.10.2.3) 1434
syncstream::syncbuf.cons (29.10.2.2) 1434
syncstream::syncbuf.members (29.10.2.4) 1435
syncstream::syncbuf.overview (29.10.2.1) 1433
syncstream::syncbuf.special (29.10.2.6) 1435
syncstream::syncbuf.virtuals (29.10.2.5) 1435
syntax (4.3) 8
syserr (19.5) 552
syserr.compare (19.5.5) 558
syserr.errcat (19.5.2) 554
syserr.errcat.derived (19.5.2.4) 555
syserr.errcat.nonvirtuals (19.5.2.3) 555
syserr.errcat.objects (19.5.2.5) 555
syserr.errcat.overview (19.5.2.1) 554
syserr.errcat.virtuals (19.5.2.2) 554
syserr.errcode (19.5.3) 555
syserr.errcode.constructors (19.5.3.1) 556
syserr.errcode.modifiers (19.5.3.3) 556
syserr.errcode.nonmembers (19.5.3.5) 557
syserr.errcode.observers (19.5.3.4) 557
syserr.errcode.overview (19.5.3.1) 555
syserr.errcondition (19.5.4) 557
syserr.errcondition.constructors (19.5.4.2) 558
syserr.errcondition.modifiers (19.5.4.3) 558
syserr.errcondition.nonmembers (19.5.4.5) 558
syserr.errcondition.observers (19.5.4.4) 558
syserr.errcondition.overview (19.5.4.1) 557
syserr.hash (19.5.6) 559
syserr.syserr (19.5.7) 559
syserr.syserr.members (19.5.7.2) 559
syncstream (29.10.2) 1433
syncstream::syncread (29.10.3) 1435
temp (Clause 13) 346
temp.alias (13.7.7) 383
temp.arg (13.4) 353
temp.arg.explicit (13.10.1) 414
temp.arg.nontype (13.4.2) 356
temp.arg.template (13.4.3) 357
temp.arg.type (13.4.1) 355
temp.class (13.7.1) 365
temp.class.order (13.7.5.2) 377
temp.class.spec (13.7.5) 374
temp.class.spec.match (13.7.5.1) 376
temp.class.spec.mfunc (13.7.5.3) 377
temp.concept (13.7.8) 384
temp.constr (13.5) 359
temp.constr.atomic (13.5.1.2) 360
temp.constr.constr (13.5.1) 359
temp.constr.decl (13.5.2) 361
temp.constr.normal (13.5.3) 362
temp.constr.op (13.5.1.1) 359
temp.constr.order (13.5.4) 363
temp.decls (13.7) 364
temp deduct (13.10.2) 415
temp.deduct.call (13.10.2.1) 419
temp.deduct.conv (13.10.2.3) 422
temp.deduct.decl (13.10.2.6) 430
temp.deduct.funcaddr (13.10.2.2) 422
temp.deduct.guide (13.7.1.2) 367
temp.deduct.partial (13.10.2.4) 423
temp.deduct.type (13.10.2.5) 424

Cross references 1702
temp.dep (13.8.2) 391
temp.dep.candidate (13.8.4.2) 398
temp.dep.constexpr (13.8.2.3) 396
temp.dep.expr (13.8.2.2) 395
temp.dep.res (13.8.4) 397
temp.dep.temp (13.8.2.4) 396
temp.dep.type (13.8.2.1) 392
temp.expl.spec (13.9.3) 409
temp.explicit (13.9.2) 406
temp.fct (13.7.6) 378
temp.fct.spec (13.10) 413
temp.friend (13.7.4) 372
temp.func.order (13.7.6.2) 380
temp.inject (13.8.5) 400
temp.inst (13.9.1) 402
temp.local (13.8.1) 388
temp.memo (13.7.2) 368
temp.memo.class (13.7.1.3) 367
temp.memo.enum (13.7.1.5) 368
temp.memo.func (13.7.1.1) 366
temp.names (13.3) 351
temp.nondep (13.8.3) 397
temp.over (13.10.3) 431
temp.over.link (13.7.6.1) 378
temp.param (13.2) 347
temp.point (13.8.4.1) 397
temp.pre (13.8) 385
temp.spec (13.9) 401
temp.static (13.7.1.4) 367
temp.variadic (13.7.3) 369
template.bitset (20.9.2) 609
template.gslice.array (26.7.7) 1205
template.gslice.array.overview (26.7.7.1) 1205
template间接.array (26.7.9) 1207
template间接.array.overview (26.7.9.1) 1207
template.mask.array (26.7.8) 1206
template.mask.array.overview (26.7.8.1) 1206
template.slice.array (26.7.5) 1203
template.slice.array.overview (26.7.5.1) 1203
template.valarray (26.7.2) 1193
template.valarray.overview (26.7.2.1) 1193
terminate (17.9.4.4) 517
terminate.handler (17.9.4.1) 516
thread (Clause 32) 1551
thread.barrier (32.8.2) 1593
thread.barrier.class (32.8.2.2) 1594
thread.condition (32.6) 1583
thread.condition.condvar (32.6.3) 1584
thread.condition.condvarany (32.6.4) 1587
thread.condition/nonmember (32.6.2) 1584
thread.condvarany.intwait (32.6.4.2) 1589
thread.condvarany.wait (32.6.4.1) 1588
thread.coord (32.8) 1592
thread.general (32.1) 1551
thread.jthread.class (32.4.3) 1562
thread.jthread.cons (32.4.3.1) 1563
thread.jthread.mem (32.4.3.2) 1564
thread.jthread.special (32.4.3.4) 1564
thread.jthread.static (32.4.3.5) 1564
thread.jthread.stop (32.4.3.3) 1564
thread.latch (32.8.1) 1592
thread.latch.class (32.8.1.2) 1592
thread.lock (32.5.4) 1573
thread.lock.algorithm (32.5.5) 1582
thread.lock.guard (32.5.4.1) 1573
thread.lock.scoped (32.5.4.2) 1574
thread.lock.shared (32.5.4.4) 1578
thread.lock.shared.cons (32.5.4.4.1) 1579
thread.lock.shared.locking (32.5.4.4.2) 1580
thread.lock.shared.mod (32.5.4.4.3) 1581
thread.lock.shared.obs (32.5.4.4.4) 1581
thread.lock.unique (32.5.4.3) 1575
thread.lock.unique.cons (32.5.4.3.1) 1576
thread.lock.unique.locking (32.5.4.3.2) 1577
thread.lock.unique.mod (32.5.4.3.3) 1578
thread.lock.unique.obs (32.5.4.3.4) 1578
threadmutex (32.5) 1565
thread.mutex.class (32.5.3.2.1) 1567
thread.mutex.recursive (32.5.3.2.2) 1568
thread.mutex.requirements (32.5.3) 1566
thread.mutex.requirements.general (32.5.3.1) 1566
thread.mutex.requirements.mutex (32.5.3.2) 1566
thread.once (32.5.6) 1582
thread.once.callonce (32.5.6.2) 1582
thread.once.onceflag (32.5.6.1) 1582
thread.req (32.2) 1551
thread.req.exception (32.2.2) 1551
thread.req.lockable (32.2.5) 1552
thread.req.lockable.basic (32.2.5.2) 1552
thread.req.lockable.general (32.2.5.1) 1552
thread.req.lockable.req (32.2.5.3) 1553
thread.req.lockable.timed (32.2.5.4) 1553
thread.req.native (32.2.3) 1551
thread.req.paramname (32.2.1) 1551
thread.req.timing (32.2.4) 1551
thread.sema (32.7) 1590
thread.sema.cnt (32.7.2) 1591
thread.sharedmutex.class (32.5.3.4.1) 1571
thread.sharedmutex.requirements (32.5.3.4) 1570
thread.sharedtimedmutex.class (32.5.3.5.1) 1572
thread.sharedtimedmutex.requirements (32.5.3.5) 1572
thread.stoptoken (32.3) 1553
thread.stoptoken.intro (32.3.1) 1553
thread.stoptoken.syn (32.3.2) 1554
thread.syn (32.4.1) 1558
thread.thread.algorithm (32.4.2.7) 1562
thread.thread.assign (32.4.2.4) 1561
thread.thread.class (32.4.2) 1558
thread.thread.constructor (32.4.2.4) 1561
thread.thread.destr (32.4.2.3) 1561
thread.thread.id (32.4.2.1) 1559
thread.thread.member (32.4.2.5) 1561
thread.thread.static (32.4.2.6) 1562
thread.thread.this (32.4.4) 1565
Symbols

!, see operator, logical negation
!=, see operator, inequality
(), see operator, function call, see declarator, function
/, see operator, division
+, see operator, unary plus, see operator, addition
++, see operator, increment
-, 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

bit-field declaration, 270
label specifier, 147
::*, see declarator, pointer-to-member
<, see operator, less than
<, see operator, less than
<=, see operator, less than or equal to
<=, see operator, left shift
<=, see operator, less than or equal to
<, see operator, equality
>, see operator, greater than
>=, see operator, greater than or equal to
>>, see operator, right shift
?;, see operator, conditional expression
[][], see operator, subscripting, see declarator, array
# operator, 449, 452
## operator, 452
#define, 449
#elif, 445
else, 446
#endif, 446
#error, see preprocessing directives, error
#include, 446, 471
#include, 446, 471
#line, see preprocessing directives, line control
#pragma, see preprocessing directives, pragma
#undef, 454, 482
%, see operator, remainder
&., see operator, address-of, see operator, bitwise
AND, see declarator, reference
&&, see operator, logical AND
-, see operator, bitwise exclusive OR
\, see backslash character
{} block statement, 147
class declaration, 247
class definition, 247
enum declaration, 211
initializer list, 192
_, see character, underscore
-_cppplus, 455
__cpp_aggregate_bases, 455
__cpp_aggregate_nsdmi, 455
__cpp_alias_templates, 455
__cpp_aligned_new, 455
__cpp_attributes, 455
__cpp_binary_literals, 455
__cpp_capture_star_this, 456
__cpp_char8_t, 456
__cpp_concepts, 456
__cpp_conditional_explicit, 456
__cpp_consteval, 456
__cpp_constexpr, 456
__cpp_constantexpr_dynamic_alloc, 456
__cpp_constantexpr_in_decltype, 456
__cpp_constinit, 456
__cpp_declarative_type, 456
__cpp_declarative_type_auto, 456
__cpp_deduction_guides, 456
__cpp_delegating_constructors, 456
__cpp_designated_initializers, 456
__cpp_enum_attribute, 456
__cpp_initializer_lists, 456
__cpp_inheriting_constructors, 456
__cpp_init_captures, 456
__cpp_initializer_lists, 456
__cpp_inline_variables, 456
__cpp_inheriting_constructors, 456
__cpp_if constexpr, 456
__cpp_if_constantexpr, 456
__cpp_if_impl_coroutine, 456
__cpp_if_impl_destroying_delete, 456
__cpp_if_impl_three_way_comparison, 456
__cpp_inheriting_constructors, 456
__cpp_init_captures, 456
__cpp_initializer_lists, 456
__cpp_inline_variables, 456
__cpp_lambdas, 456
__cpp_lines, 456
__cpp_modules, 456
__cpp_namespace_attribute, 456
__cpp_noexcept_function_type, 456
__cpp_nonetype_template_arg, 456
__cpp_nonetype_template_parameter_auto, 456
© ISO/IEC  N4861

__cpp_nsdmi, 456
__cpp_range_based_for, 456
__cpp_raw_strings, 456
__cpp_ref_qualifiers, 456
__cpp_return_type_deduction, 456
__cpp_rvalue_references, 456
__cpp_structured_deallocation, 456
__cpp_static_assert, 456
__cpp_unicode_characters, 456
__cpp_unicode_literals, 456
__cpp_user_defined_literals, 456
__cpp_using_enum, 456
__cpp_variable_templates, 456
__cpp_variadic_templates, 456
__cpp_variadic_using, 456
__DATE__, 455
__FILE__, 455
__has_cpp_attribute, 444
__has_include, 444
__LINE__, 455
__STDC__, 457
__STDC_HOSTED__, 455
__STDC_ISO_10646__, 457
__STDC_MB_MIGHT_NEQ_WC__, 457
__STDC_VERSION__, 457
__STDCPP_DEFAULT_NEW_ALIGNMENT__, 455
__STDCPP_DEFAULT_NEW_ALIGNMENT__, 455
__STDCPP_THREADSAFE_STATIC_INIT, 456
__STDCPP_THREADS__, 457
__TIME__, 455
__VA_ARGS__, 449, 450
__VA_OPT__, 449–451
1, see operator, bitwise inclusive OR
11, see operator, logical OR
\&, see operator, ones’ complement, see destructor

Numbers
0, see also zero, null
  null character, see character, null
  string terminator, 23

A
abbreviated
  template function, see template, function,
  abbreviated
abort, 85, 152
absolute path, 1443
abstract class, see class, abstract
abstract-declarator, 176, 1622
abstract-pack-declarator, 176, 1623
access, 3
access control, 285–294
  anonymous union, 274
  base class, 288
  base class member, 275
class member, 113
default, 286
default argument, 287
friend function, 290
member function and, 257
member name, 286
multiple access, 294
nested class, 294
overload resolution and, 284
overloading and, 314
protected, 285
public, 285
using-declaration and, 225
virtual function, 293

access specifier, 287, 288
access-specifier, 275, 1627
accessible, 289
active
  union member, 272
active macro directive, see macro, active
addition operator, see operator, addition
additive-expression, 132, 1617
address, 72, 135
addressable function, 480
aggregate, 192
elements, 193
aggregate deduction candidate, see candidate, aggregate deduction
aggregate initialization, 192
algorithm
  stable, 462, 486
<algorithm>, 492, 494, 1029, 1647
alias
  namespace, 218
alias template, see template, alias
alias-declaration, 156, 1620
alignas, 16, 229, 230, 1625
alignment, 64
  extended, 65
  fundamental, 64
  new-extended, 65
  stricter, 65
  stronger, 65
  weaker, 65
alignment requirement
  implementation-defined, 64
alignment-specifier, 229, 1625
alignof, 16, 120, 123, 395, 396, 1617
allocated type, see type, allocated
allocation
  alignment type, see type, allocated
  allocation storage, 126
  implementation-defined bit-field, 270
  unspecified, 253
allocation functions, 62
alternate form
  format string, 725
alternative token, see token, alternative
ambiguity
  base class member, 283
class conversion, 285
declaration type, 158
declaration versus cast, 176
declaration versus expression, 154
function declaration, 190
member access, 283
overloaded function, 314
parentheses and, 124
ambiguous conversion sequence, see conversion sequence, ambiguous
Amendment 1, 482
and, 16
and-expression, 136, 1618
and_eq, 16
anonymous union, 273
anonymous union object, 273<br/>&lt;any&gt;, 492, 604, 1642
appearance-ordered, 83
appertain, 229
argc, 82
argument, 3, 484, 485, 549
access checking and default, 287
binding of default, 187
evaluation of default, 187
example of default, 186, 187
function call expression, 3
function-like macro, 3
overloaded operator and default, 339
reference, 112
scope of default, 187
template, 353
template instantiation, 3
throw expression, 3
type checking of default, 187
argument and name hiding
default, 187
argument and virtual function
default, 188
argument forwarding call wrapper, 669
argument list
empty, 183
variable, 183
argument passing, 112
reference and, 197
argument substitution, see macro, argument substitution
argument type
unknown, 183
argv, 82
arithmetic
pointe, 132
unsigned, 70
array
bound, 181
const, 73
delete, 128
element, 181
handler of type, 436
new, 125
overloading and pointer versus, 312
parameter of type, 183
sizeof, 123
template parameter of type, 349<br/>&lt;array&gt;, 492–494, 579, 818, 821, 957, 1647, 1672, 1673
array
as aggregate, 821
contiguous storage, 821
creation, 823
initialization, 821, 822
tuple interface to, 824
zero sized, 823
array size
default, 181
array type, 181
arrow operator, see operator, class member access
as-if rule, 7
asm, 16, 226, 1625
implementation-defined, 226
asm-declaration, 226, 1625
assembler, 226<br/>&lt;assert.h&gt;, 471, 550, 1659, 1663
assignment
and lvalue, 139
conversion by, 139
copy, see assignment operator, copy
move, see assignment operator, move, 460
reference, 197
assignment operator
copy, 256, 261–263
hidden, 262
implicitly declared, 261
implicitly defined, 263
non-trivial, 263
trivial, 263
virtual bases and, 263
move, 256, 261–263
hidden, 262
implicitly declared, 262
implicitly defined, 263
non-trivial, 263
trivial, 263
overloaded, 339
assignment-expression, 139, 1618
assignment-operator, 139, 1618
associated, 1553
associated constraints, 361
associative containers
exception safety, 807
requirements, 807
unordered, see unordered associative containers
asynchronous provider, 1597
asynchronous return object, 1597
at least as constrained, 363
at least as specialized as, see more specialized
atexit, 85
atomic
notifying operation, 1526
operation, 77–81
smart pointers, 1543–1548
waiting operation, 1526
eligible to be unblocked, 1527
<bratomic>, 470, 492, 1520, 1647
atomic constraint, 360
identical, 360
<bratomics>, 1682
attached
  declaration, 237
  entity, 53
attribute, 228–232
  alignment, 230
  carries dependency, 231
  deprecated, 232
  fallback, 232
  likely, 233
  maybe unused, 233
  no unique address, 235
  nodiscard, 234
  noreturn, 235
  syntax and semantics, 228
unlikely, 233
attribute, 229, 1625
attribute-argument-clause, 229, 1625
attribute-declaration, 156, 1620
attribute-list, 229, 1625
attribute-namespace, 229, 1625
attribute-scoped-token, 229, 1625
attribute-specifier, 229, 1625
attribute-specifier-list, 229, 1625
attribute-specifier-seq, 229, 1625
attribute-token, 229, 1625
auto, 16, 151, 170, 1622
automatic storage duration,  see storage duration, automatic
await-expression, 121, 1617
awk, 1493

B
backslash character, 19
bad_alloc, 127
bad_cast, 115
bad_typeid, 115
balanced-token, 229, 1625
balanced-token-seq, 229, 1625
barrier
  phase synchronization point, 1594
<brbarrier>, 492, 1593, 1638
barrier phase, 1594
base characteristic, 689
base class, 275, 276
  dependent, 393
  direct, 275
  indirect, 275
  non-virtual, 277
  overloading and, 313
  private, 288
  protected, 288
  public, 288
  virtual, 277
base class subobject, 55
base prefix, 727
base-2 representation, 71
base-clause, 275, 1627
base-specifier, 275, 1627
base-specifier-list, 275, 1627
behavior
  conditionally-supported, 3, 7
  default, 460, 464
  implementation-defined, 4, 8
  locale-specific, 4
  observable, 7, 8
  on receipt of signal, 76
  required, 461, 464
  undefined, 5, 7, 8, 955
  unspecified, 5, 8
Ben, 313
Bernoulli distributions, 1175–1177
bernoulli_distribution
  discrete probability function, 1175
Bessel functions
  I_ν, 1220
  J_ν, 1220
  K_ν, 1220
  N_ν, 1221
  j_ν, 1223
  n_ν, 1223
beta functions B, 1219
better conversion,  see conversion, better
better conversion sequence,  see conversion sequence, better
binary fold, 107
binary left fold, 107
binary operator
  interpretation of, 339
  overloaded, 339
binary operator function,  see operator function, binary
binary right fold, 107
binary-digit, 17, 1612
binary-exponent-part, 20, 1613
binary-literal, 17, 1612
bind directly, 199
binding
  reference, 197
binomial_distribution
  discrete probability function, 1175
<brbit>, 492, 493, 1150, 1638
bit-field, 270
  address of, 270
  alignment of, 270
  implementation-defined alignment of, 270
  implementation-defined sign of, 1657
  type of, 270
  unnamed, 270
Index

bitand, 16

<bitset>, 609

block (execution), 3, 1553, 1561, 1564–1570, 1585, 1588, 1589, 1591–1595, 1602, 1604, 1605
  with forward progress guarantee delegation, 81

block (statement), 3, see statement, compound initialization in, 153
  scope, 35
  structure, 153

<body>

Bond
  James Bond, 104

bool, 16, 167, 1621

Boolean literal, 23
  see literal, boolean

Boolean type, 71
  boolean-literal, 23, 1614
  bound argument entity, 668
  bound arguments, 681
  bound, of array, 181
  brace-or-equal-initializer, 189, 1623
  braced-init-list, 189, 1623
  brains
    names that want to eat your, 481

break, 16, 152, 1619

buckets, 808

built-in candidate, 318

built-in operators, see operators, built-in

byte, 54, 123

C

C
  linkage to, 226
  standard, 1
  standard library, 2
  c-char, 18, 1613
  c-char-sequence, 18, 1613

C++ library headers
  importable, 468

call
  nodiscard, 234
  operator function, 338

call pattern, 669

call signature, 668

call wrapper, 668, 669
  forwarding, 669

C

C
  cast
    base class, 117
    const, 119, 130
    derived class, 117
    dynamic, 114, 512
      construction and, 303
      destruction and, 303
    integer to pointer, 118
    lvalue, 116, 118
    pointer to integer, 118
    pointer-to-function, 118
    pointer-to-member, 117, 119
    reference, 116, 119
    reinterpret, 118, 130
      integer to pointer, 118
      lvalue, 118
    pointer to integer, 118
    pointer-to-function, 118
    pointer-to-member, 119
    reference, 119
    static, 116, 130
    lvalue, 116
    reference, 116
  undefined pointer-to-function, 118
  cast-expression, 130, 1617
  casting, 113
  casting away constness, 119
  catch, 16, 433
  category tag, 928
  cats
    interfering with canines, 511
  cauchy_distribution
    probability density function, 1183
  <cassert>, 471, 550, 1659
  <cassert>, 471, 550, 1659
  cast
© ISO/IEC

Index 1712

<N4861>
complete object of, 56
complete-class context, 252
completely defined, 252
<complex>, 492, 1142, 1143, 1638, 1663
<complex.h>, 1659, 1663, 1664
component, 459
composite pointer type, 88
compound-requirement, 109, 1616
compound-statement, 147, 1619
concatenation
macro argument, see ## operator
string, 22
concept, 384
  model, 484
type, 384
concept, 16, 384, 1628
concept-definition, 384, 1628
concept-id, 353
concept-name, 384, 1628
<concepts>, 492, 535, 1638
concurrent forward progress guarantees, 81
condition, 146, 1619
conditions
rules for, 146
<condition_variable>, 1583, 1647
conditional-expression
  throw-expression in, 137
conditional-expression, 136, 1618
conditionally-supported behavior, see behavior,
  conditionally-supported
conditionally-supported-directive, 442, 1630
conflict, 77
conformance requirements, 7–8
  class templates, 7
classes, 7
genral, 7
library, 7
  method of description, 7
conjunction, 359
consistency
  linkage, 159
  linkage specification, 228
type declaration, 53
const, 16, 73, 175, 1622
  cast away, 119
  constructor and, 256, 258
  destructor and, 256, 264
  linkage of, 51
  overloading and, 312
const member function, 256
const object, see object, const
  undefined change to, 166
const volatile member function, 256
const volatile object, see object, const volatile
const-default-constructible, 190
const-qualified, 73
const-volatile-qualified, 73
const_cast, 16, see cast, const, 110, 395, 396, 1616
const_local_iterator, 809
constant, 16, 95
  enumeration, 211
null pointer, 93, 94
constant destruction, see destruction, constant
constant expression, 140, see expression, constant
  permitted result of, 144
constant initialization, 82
constant iterator, 908
constant subexpression, 459
constant-expression, 140, 1618
constant-initialized, 140
constexpr, 16, 158, 1620
constexpr function, 163
constexpr if, 148
constexpr iterators, 909
constexpr-compatible
  defaulted comparison operator, 306
  defaulted special member function, 257
constinit, 16, 158, 164, 165, 1620, 1633
constitu member function, 74
constraint, 359
  associated, see associated constraints
  immediately-declared, 348
  normalization, 362–363
  satisfaction
  atomic, 361
  conjunction, 359
  disjunction, 359
  subsumption, 363
constraint-expression, 361, 1628
constraint-logical-and-expression, 346, 1628
constraint-logical-or-expression, 346, 1628
construction, 301–303
  dynamic cast and, 303
  member access, 301
  move, 460
  pointer to member or base, 302
  typeid operator, 303
  virtual function call, 302
constructor, 257
  address of, 258
  array of class objects and, 295
  converting, 267
  copy, 66, 256, 259–261, 467
  elision, 303
  implicitly declared, 260
  implicitly defined, 261
  nontrivial, 260
  trivial, 260
default, 256, 258
  non-trivial, 258
  trivial, 258
exception handling, see exception handling,
  constructors and destructors
explicit call, 257
  implicitly called, 259
  implicitly defined, 258
inheritance of, 258
inherited, 221
move, 256, 259–261
elision, 303
implicitly declared, 260
implicitly defined, 261
non-trivial, 260
trivial, 260
non-trivial, 258
random number distribution requirement, 1161
random number engine requirement, 1158
union, 272
constructor, conversion by, see conversion, user-defined
contained value
any, 605
optional, 583
variant, 595
container
contiguous, 789
contains a value
optional, 583
context
non-deduced, 425
contextually converted constant expression of type
bool, see conversion, contextual
contextually converted to bool, see conversion, contextual
contextually implicitly converted, 90
contiguous container, 789
continue, 16, 152, 1619
and handler, 433
and try block, 433
control line, see preprocessing directive
control-line, 442, 1629
conventions, 464
lexical, 10–25
conversion
argument, 183
array-to-pointer, 91
better, 334
bool, 93
boolean, 94
class, 266
contextual, 90
contextual to bool, 90
contextual to constant expression of type
bool, 144
deduced return type of user-defined, 269
derived-to-base, 329
floating to integral, 93
floating-point, 93
function pointer, 94
function-to-pointer, 91
implementation-defined pointer integer, 118
implicit, 89, 90, 266
implicit user-defined, 266
inheritance of user-defined, 268
integer rank, 74
integral, 93
integral to floating, 93
lvalue-to-rvalue, 90, 1653
narrowing, 203
null member pointer, 94
null pointer, 93
overload resolution and, 326
overload resolution and pointer, 338
pointer, 93
pointer-to-member, 94
void*, 94
qualification, 91–92
return type, 152
standard, 89–94
temporary materialization, 91
to signed, 93
to unsigned, 93
type of, 268
user-defined, 266, 267
usual arithmetic, 94
virtual user-defined, 268
conversion explicit type, see casting
conversion function, see conversion, user-defined, 268
conversion rank, 330
conversion sequence
ambiguous, 330
better, 334
implicit, 328
indistinguishable, 334
standard, 89
user-defined, 331
worse, 334
conversion-declarator, 268, 1627
conversion-function-id, 267, 1627
conversion-type-id, 267, 1627
converted constant expression, see expression, converted constant
converting constructor, see constructor, converting copy
class object, see constructor, copy, see assignment operator, copy
copy constructor
random number engine requirement, 1158
copy deduction candidate, 322
copy elision, see constructor, copy, elision
copy-initialization, 191
copy-list-initialization, 199
core constant expression, see expression, core constant
coroutine, 207
promise type, 207
resumer, 208
<coroutine>, 493, 527, 528, 1638
coroutine return, see co_return
coroutine state, 208
coroutine-return-statement, 153, 1619
counted range, see range, counted
**Index**

<table>
<thead>
<tr>
<th>Cpp17Allocator</th>
<th>475</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cpp17BinaryTypeTrait</td>
<td>689</td>
</tr>
<tr>
<td>Cpp17Clock</td>
<td>1239</td>
</tr>
<tr>
<td>Cpp17CopyAssignable</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17CopyConstructible</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17CopyInsertable into X</td>
<td>789</td>
</tr>
<tr>
<td>Cpp17DefaultConstructible</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17DefaultInsertable into X</td>
<td>789</td>
</tr>
<tr>
<td>Cpp17Destructible</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17EmplaceConstructible into X from args</td>
<td>789</td>
</tr>
<tr>
<td>Cpp17EqualityComparable</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17Erasable from X</td>
<td>790</td>
</tr>
<tr>
<td>Cpp17Hash</td>
<td>474</td>
</tr>
<tr>
<td>Cpp17LessThanComparable</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17MoveAssignable</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17MoveConstructible</td>
<td>471</td>
</tr>
<tr>
<td>Cpp17MoveInsertable into X</td>
<td>789</td>
</tr>
<tr>
<td>Cpp17NullablePointer</td>
<td>474</td>
</tr>
<tr>
<td>Cpp17TransformationTrait</td>
<td>689</td>
</tr>
<tr>
<td>Cpp17UnaryTypeTrait</td>
<td>689</td>
</tr>
<tr>
<td>&lt;csetjmp&gt;</td>
<td>482, 532, 1660</td>
</tr>
<tr>
<td>&lt;csignal&gt;</td>
<td>532, 533</td>
</tr>
<tr>
<td>&lt;cstdlib&gt;</td>
<td>85, 470, 490, 503, 504, 507, 532, 614, 630, 783, 1140, 1190, 1217, 1334, 1659, 1660, 1664</td>
</tr>
<tr>
<td>&lt;cstring&gt;</td>
<td>255, 467, 781, 1659, 1667, 1670</td>
</tr>
<tr>
<td>&lt;ctgmath&gt;</td>
<td>absence thereof, 482, 1638</td>
</tr>
<tr>
<td>&lt;cstdio&gt;</td>
<td>504, 1358, 1359, 1365, 1424, 1482, 1483, 1659</td>
</tr>
<tr>
<td>&lt;cstdlib&gt;</td>
<td>85, 470, 490, 503, 504, 507, 532, 614, 630, 783, 1140, 1190, 1217, 1334, 1659, 1660, 1664</td>
</tr>
<tr>
<td>&lt;cstring&gt;</td>
<td>255, 467, 781, 1659, 1667, 1670</td>
</tr>
<tr>
<td>&lt;ctgmath&gt;</td>
<td>abundance thereof, 482, 1638</td>
</tr>
<tr>
<td>&lt;ctime&gt;</td>
<td>1313, 1316, 1659</td>
</tr>
<tr>
<td>&lt;ctypes.h&gt;</td>
<td>780, 1663</td>
</tr>
<tr>
<td>&lt;cstdint&gt;</td>
<td>482, 783, 1647, 1659</td>
</tr>
<tr>
<td>current instantiation, 392</td>
<td></td>
</tr>
<tr>
<td>dependent member of the, 393</td>
<td></td>
</tr>
<tr>
<td>member of the, 393</td>
<td></td>
</tr>
<tr>
<td>currently handled exception, see exception handling, currently handled exception customization point, 480</td>
<td></td>
</tr>
<tr>
<td>cv-combined type, see type, cv-combined</td>
<td></td>
</tr>
<tr>
<td>cv-decomposition, 91</td>
<td></td>
</tr>
<tr>
<td>cv-qualification signature, 92</td>
<td></td>
</tr>
<tr>
<td>cv-qualifier, 73</td>
<td></td>
</tr>
<tr>
<td>top-level, 73</td>
<td></td>
</tr>
<tr>
<td>cv-qualifier-seq, 175, 1622</td>
<td></td>
</tr>
<tr>
<td>function, see conversion, function-to-pointer</td>
<td></td>
</tr>
<tr>
<td>&lt;cwchar&gt;, 482, 742, 781–783, 1659</td>
<td></td>
</tr>
<tr>
<td>&lt;cwctype&gt;, 482, 780</td>
<td></td>
</tr>
</tbody>
</table>

**D**

| d-char | 21, 1614 |
| d-char-sequence | 21, 1614 |
| DAG |
| multiple inheritance, 277, 278 |
| non-virtual base class, 278 |
| virtual base class, 277, 278 |
| data member, see member, 251 |
| static, 251 |
| data race, 79 |
| deadlock, 460 |
| deallocation function |
| usual, 63 |
| deallocation functions, 62 |
| decay |
| array, see conversion, array-to-pointer |
| function, see conversion, function-to-pointer |
| decimal-floating-point-literal | 20, 1613 |
| decimal-literal | 17, 1612 |
| decl-reachable, 241 |
| decl-specifier | 158, 1620 |
| decl-specifier-seq | 158, 1620 |
| declaration, 26, 156–232 |
| array, 181 |
| asm, 226 |
| bit-field, 270 |
| class name, 27 |
| constant pointer, 178 |
| default argument, 186–188 |
| definition versus, 26 |
| disqualifying, 543 |
| ellipsis in function, 112, 183 |
| enumerator point of, 34 |
| exported, 237 |
| extern | 27 |
| extern reference, 197 |
| forward, 159 |
| forward class, 250 |
| function, 26, 157, 182 |
| local class, 274 |
| member, 250 |
| multiple, 53 |
| name, 26 |
| object, 157 |
| opaque enum, 27 |
| overloaded, 311 |
| overloaded name and friend, 291 |
| parameter, 27, 183 |
| parentheses in, 176, 178 |
| point of, 33–34 |
| pointer, 178 |
| reference, 179 |
| static member, 27 |
| storage class, 158 |
| structured binding, see structured binding declaration type, 177 |
| typedef, 157 |
typedef, 27
datatype, 27 as type, 160
declaration, 156, 1620
declaration hiding, see name hiding
declaration-seq, 156, 1619
declaration-statement, 153, 1619
deaferative region, 32
declarator, 27, 157, 174–204
array, 181
function, 182–186
meaning of, 177–188
multidimensional array, 182
pointer, 178
pointer-to-member, 180
reference, 179
declarator, 175, 1622
declarator-id, 175, 1622
declared specialization, see specialization, declared
decltype, 16, 169, 170, 1621, 1622
decltype-specifier, 169, 1621
decreemnt operator
overloaded, see overloading, decreemnt;
operator
decreemnt operator function, see operator
function, decreemnt
deducible template, see template, deducible
deduction
class template argument, 367
class template arguments, 113, 167, 173, 322
placeholder type, 172
deduction-guide, 367, 1628
default, 16, 147, 204, 1619, 1623
default access control, see access control, default
default argument
overload resolution and, 326
default argument instantiation, 405
default constructor, see constructor, default
random number distribution requirement, 1161
seed sequence requirement, 1157
default initializers
overloading and, 313
default member initializer, 252
default memory resource pointer, 657
default-initialization, 189
default-inserted, 789
defaulted, 206
deferred function, 1606
define, 26
defined, 444
defined-macro-expression, 444, 1630
defining-type-id, 176, 1622
defining-type-specifier, 165, 1621
defining-type-specifier-seq, 165, 1621
definition, 26, 27
alternate, 483
class, 247, 251
class name as type, 249
constructor, 204
coroutine, 207
declaration as, 157
declaration-statement, 153, 1619
function, 204–207
deleted, 206
explicitly-defaulted, 205
local class, 274
member function, 254
namespace, 214
nested class, 271
program semantics affected by, 404
pure virtual function, 282
scope of class, 249
static member, 269
virtual function, 280
definition domain, 30
definitions, 3–6
delete
array, 128
single-object, 128
delete, 16, 62, 128, 204, 308, 339, 395, 1617, 1623
destructor and, 129, 265
operator
replaceable, 483
overloading and, 63
single-object, 128
type of, 309
undefined, 128
delete-expression, 128, 1617
deleter, 630
denormalized value, see number, subnormal
dependency-ordered before, 78
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>, 492, 493, 818, 819, 957
dereferenceable iterator, see iterator,
dereferenceable
dereferencing, see indirection
derivation, see inheritance
derived class, 275–283
most, see most derived class
derived object
most, see most derived object
designated-initializer-clause, 189, 1623
designated-initializer-list, 189, 1623
designator, 189, 1623
destringization, 457
destroying operator delete, see operator delete,
destroying
destruction, 301–303
constant, 143
dynamic cast and, 303
member access, 301
pointer to member or base, 302
virtual function call, 302
destructor, 263, 264, 467
default, 264
exception handling, see exception handling, constructors and destructors
explicit call, 265
implicit call, 265
implicitly defined, 264
non-trivial, 264
program termination and, 265
prospective, 263
pure virtual, 264
selected, 264
union, 272
virtual, 264
diagonosable rules, 7
diagnostic message, see message, diagnostic
difference type, 908
digit, 14, 1611
digit-sequence, 20, 1613
digraph, see token, alternative, 13
direct base class, see base class, direct
direct member, 251
direct-initialization, 191
direct-list-initialization, 199
direct-non-list-initialization, 460
directed acyclic graph, see DAG
directive, preprocessing, see preprocessing
directive-introducing token, see token, directive-introducing
directory, 1437
directory-separator, 1446
discard
    random number engine requirement, 1158
discard_block_engine
generation algorithm, 1167
state, 1166
textual representation, 1167
transition algorithm, 1167
discarded
    declaration, 242
discarded statement, 148
discarded-value expression, 89
discrete probability function
    bernoulli_distribution, 1175
    binomial_distribution, 1175
    discrete_distribution, 1185
    geometric_distribution, 1176
    negative_binomial_distribution, 1177
    poisson_distribution, 1177
    uniform_int_distribution, 1173
discrete_distribution
    discrete probability function, 1185
    weights, 1185
disjunction, 359
disqualifying declaration, see declaration, disqualifying
disqualifying parameter, see parameter, disqualifying
distribution, see random number distribution
do, 16, 149, 1619
dogs
    obliviousness to interference, 511
domain error, 1218
dominance
    virtual base class, 284
dot
    filename, 1446
dot operator, see operator, class member access
dot-dot
    filename, 1446
double, 16, 21, 167, 1621
dynamic binding, see function, virtual
dynamic initialization, 82
dynamic type, see type, dynamic
dynamic_cast, 16, see cast, dynamic, 110, 395, 1616
E
    E (complete elliptic integrals), 1219
    E (incomplete elliptic integrals), 1221
    ECMA-262, 2
    ECMAScript, 1493, 1517
    egrep, 1493
    Ei (exponential integrals), 1222
    elaborated type specifier, see class name, elaborated
    elaborated-enum-specifier, 168, 1621
    elaborated-type-specifier, 168, 1621
    element access functions, 1026
element type, 181
eif-group, 442, 1630
eif-groups, 442, 1630
eligible special member function, see special member function, eligible
eligible to be unblocked, 1527
elision
    copy, see constructor, copy, elision
    copy constructor, see constructor, copy, elision
    move constructor, see constructor, move, elision
ellipsis
    conversion sequence, 112, 331
    overload resolution and, 326
elliptic integrals
    complete Π, 1220
    complete E, 1219
    complete K, 1219
    incomplete Π, 1221
    incomplete E, 1221
    incomplete F, 1221
else, 16, 147, 148, 1619
else-group, 442, 1630
empty-declaration, 156, 1620
enclosing namespace set, 215
enclosing statement, 146
enclosing-name-space-specifier, 214, 1624
encoded character type, 1438
encoding
    multibyte, 23
encoding-prefix, 18, 1612
end-of-file, 614
endif-line, 442, 1630
engine, see random number engine
engine adaptor, see random number engine adaptor
engines with predefined parameters
default_random_engine, 1170
knuth_b, 1170
minstd_rand, 1169
minstd_rand0, 1169
mt19937, 1169
mt19937_64, 1169
ranlux24, 1170
ranlux24_base, 1170
ranlux48, 1170
ranlux48_base, 1170
entity, 26
    associated, 42
    implicitly movable, 304
    local, 26
    templated, 347
enum, 16, 72, 168, 211, 1621, 1624
    overloading and, 312
    type of, 211, 212
    underlying type, see type, underlying
enum name
typedef, 162
enum-base, 211, 1624
enum-head, 211, 1624
enum-head-name, 211, 1624
enum-key, 211, 1624
enum-name, 211, 1623
enum-specifier, 211, 1623
enumerated element, 466
enumerated type, see type, enumerated
enumeration, 211
    linkage of, 51
    scoped, 211
    unscoped, 211
    using declaration, 213
enumeration scope, 37
enumeration type
    conversion to, 117
    static_cast
        conversion to, 117
enumerator
definition, 28
    scoped, 211
    unscoped, 211
    value of, 211
enumerator, 211, 1624
enumerator-definition, 211, 1624
enumerator-list, 211, 1624
environment
    program, 82
epoch, 1239
equality operator function, see operator function, equality
equality-expression, 134, 1618
equivalence
    template type, 364
type, 160, 249
equivalent
    expressions, 379
    function templates, 380
    functionally, see functionally equivalent
template-heads, 380
template-parameters, 380
equivalent parameter declarations, 312
    overloading and, 312
equivalent-key group, 807
equivalently-valued, 478
<errno.h>, 550, 1663
escape character, see backslash character
escape sequence
format string, 723
    undefined, 19
escape-sequence, 19, 1613
Eulerian integral of the first kind, see beta
evaluation, 75
    order of argument, 112
    signal-safe, 533
    unspecified order of, 76, 83
    unspecified order of argument, 112
    unspecified order of function call, 112
exception
    arithmetic, 86
    undefined arithmetic, 86
<exception>, 494, 515
exception handling, 433–441
    constructors and destructors, 435
    currently handled exception, 437
    exception object, 434, 435
        constructor, 435
        destructor, 435
        function try block, 433
goto, 433
    handler, 433, 434, 436–437, 487
        active, 437
        array in, 436
        incomplete type in, 436
        match, 436–437
        pointer to function in, 436
        rvalue reference in, 436
memory, 435
    nearest handler, 434
    rthrow, 139, 435
    rethrowing, 435
switch, 433
terminate called, 139, 435, 438
throwing, 138, 434
try block, 433
exception object, see exception handling,
exception object
exception specification, 437–440
  noexcept
    constant expression and, 438
    non-throwing, 438
    potentially-throwing, 437
    virtual function and, 438
exception-declaration, 433, 1629
exclusive-or-expression, 136, 1618
<execution>, 493, 717, 718, 1642
execution agent, 1552
execution policy, 717
execution step, 80
exit, 82, 84, 152
explicit, 16, 160, 1621
explicit type conversion, see casting
explicit-instantiation, 406, 1628
explicit-specialization, 409, 1629
explicitSpecifier, 160, 1621
explicitly captured, 103
explicitly initialized elements
  aggregate, 193
exponent-part, 20, 1613
exponential integrals Ei, 1222
exponential_distribution
  probability density function, 1178
export, 16, 237, 443, 447, 1625, 1630
export-declaration, 237, 1625
exposure, 53
expr-or-braced-init-list, 189, 1623
expression, 86–145
  additive operators, 132
  alignof, 123
  assignment and compound assignment, 139
  await, 121
  bitwise AND, 135
  bitwise exclusive OR, 136
  bitwise inclusive OR, 136
  cast, 113, 130–131
  class member access, 113
  comma, 140
  conditional operator, 136
  const cast, 119
  constant, 140, 144
  converted constant, 143
  core constant, 141
  decrement, 114, 121
  delete, 128
  destructor call, 98
  dynamic cast, 114
  equality operators, 134
  equality-preserving, 534
  equivalent, see equivalent, expressions
    fold, 107
    function call, 111
    functionally equivalent, see functionally
equivalent, expressions
    increment, 114, 121
    integral constant, 143
    lambda, 98–107
  left-shift-operator, 133
  logical AND, 136
  logical OR, 136
  multiplicative operators, 131
  new, 124
  noexcept, 123
  order of evaluation of, 86
  parenthesized, 96
  pointer-to-member, 131
  pointer-to-member constant, 120
  postfix, 110–120
  potentially constant evaluated, 145
  potentially evaluated, 28
  primary, 95–110
  pseudo-destructor call, 98
  reference, 88
  reinterpret cast, 118
  relational operators, 134
  requires, 107–110
  right-shift-operator, 133
  rvalue reference, 87
sizeof, 123
  spaceship, 133
  static cast, 116
  three-way comparison, 133
throw, 138
type identification, 115
type-dependent, 391
  unary, 120–124
  unary operator, 120
  value-dependent, 391
  yield, 138
expression, 140, 1618
expression-equivalent, 460
expression-list, 111, 1616
expression-statement, 147, 1619
extend, see namespace, extend
  extended alignment, 65
  extended integer type, 71
  extended signed integer type, 70
  extended unsigned integer type, 70
extern, 16, 158, 226, 406, 1620, 1625, 1628
  linkage of, 159
extern "C", 471, 482
extern "C++", 471, 482
extern template, see instantiation, explicit
  external linkage, 51
extreme_value_distribution
  probability density function, 1180
F
F (incomplete elliptic integrals), 1221
  facet, 1319
fallback-separator, 1446
false, 16
<fenv.h>, 1142, 1663
file, 1437
  file attributes, 1463
cached, 1463
file system, 1437
file system race, 1438
file, source, see source file
filename, 1446
filename, 1446
<filesystem>, 492, 493, 1439, 1642
final, 15, 247, 251, 1626
final overrider, 278
final suspend point, 208
finite state machine, 1486
fisher_f_distribution
probability density function, 1184
float, 16, 21, 167, 1621
<float.h>, 502, 1663
floating-point literal, see literal, floating-point
floating-point promotion, 93
floating-point type, see type, floating-point
implementation-defined, 71
floating-point-literal, 20, 1613
floating-point-suffix, 20, 1613
fold
binary, 107
unary, 107
fold-expression, 107, 1616
fold-operator, 107, 1616
for, 16, 149–151, 1619
scope of declaration in, 151
for-range-declaration, 149, 1619
for-range-initializer, 149, 1619
=format>, 493, 721, 1638
format specification
format string, 724
format specifier, 1486
format string, 723
Formatter, 730
forward, 564
forward progress guarantees
concurrent, 81
delegation of, 81
parallel, 81
weakly parallel, 81
<forward_list>, 492, 493, 818, 820, 957, 1647
forwarding reference, 420
fractional-constant, 20, 1613
free store, see also new, delete, 308
freestanding implementation, 7
friend
virtual and, 280
access specifier and, 292
class access and, 290
inheritance and, 292
local class and, 292
template and, 372
friend, 16, 158, 169, 291, 1620
friend function
access and, 290
inline, 291
linkage of, 291
member function and, 290
nested class, 272
<fstream>, 1421, 1481
full-expression, 75
function, see also friend function; member
function; inline function; virtual function
allocation, 62, 125
comparison, 459
conversion, 267
deallocation, 63, 308
definition, 28
global, 482, 485
handler, 460
handler of type, 436
inline, 165
linkage specification overloaded, 228
modifier, 460
named by expression or conversion, 29
needed for constant evaluation, 145
observer, 461
operator, 338
template, 338
overload resolution and, 315
overloaded, see overloading
overloading and pointer versus, 312
parameter of type, 183
pointer to member, 131
program semantics affected by the existence
of a function definition, 404
replacement, 461
reserved, 462
template parameter of type, 349
viable, 314
virtual, 278–282
pure, 282
virtual function call, 111
virtual member, 483
function argument, see argument
function call, 112
recursive, 113
undefined, 118
function call operator
overloaded, 340
function call operator function, see operator
function, function call
function object, 666
binders, 680–682
mem_fn, 682
reference_wrapper, 669
type, 666
wrapper, 682–685
function parameter, see parameter
function parameter pack, 370
function parameter scope, 35
function pointer type, 72
function return, see return
function return type, see return type
function try block, see exception handling,
function try block
Index

H

H_n (Hermite polynomials), 1222
h-char, 14, 1611
h-char-sequence, 14, 1611
h-pp-tokens, 444, 1630
h-preprocessing-token, 444, 1630
handler, see exception handling, handler
handler, 433, 1629
handler-seq, 433, 1629
happens after, 78
happens before, 78
hard link, 1437
has-attribute-expression, 444, 1630
has-include-expression, 444, 1630
hash
instantiation restrictions, 688
hash code, 808
hash function, 807
hash tables, see unordered associative containers
header, 468
C, 482, 485, 1664
C library, 471
C++ library, 468
importable, 240
name, 14
header unit, 240
preprocessing, 447
header-name, 14, 1611
header-name-tokens, 444, 1630
headers
C library, 1663
heap with respect to comp and proj, 1112
Hermite polynomials H_n, 1222
hex-quad, 11, 1610
hexadecimal-digit, 17, 1612
hexadecimal-digit-sequence, 17, 1612
hexadecimal-escape-sequence, 19, 1613
hexadecimal-floating-point-literal, 20, 1613
hexadecimal-fractional-constant, 20, 1613
hexadecimal-literal, 17, 1612
hexadecimal-prefix, 17, 1612
hiding, see name hiding
high-order bit, 54
hosted implementation, 7

I

I_ν (Bessel functions), 1220
id
qualified, 97, 98
id-expression, 96
id-expression, 96, 1615
identical
atomic constraints, see atomic constraint,
identical
identifier, 14–15, 97, 157
identifier, 14, 1611
identifier label, 147
identifier-list, 443, 1630
identifier-nondigit, 14, 1611
if, 16, 147, 148, 150, 1619
if-group, 442, 1629
if-section, 442, 1629
ill-formed program, see program, ill-formed
immediate function, 162
immediate function context, 144
immediate invocation, 144
immediate subexpression, 74
implementation
freenstanding, 470
hosted, 470
implementation limits, see limits, implementation
implementation-defined behavior, see behavior,
implementation-defined
implementation-dependent, 408, 492, 1387, 1398
implementation-generated, 28
implicit conversion, see conversion, implicit
implicit conversion sequence, see conversion
sequence, implicit
implicit object parameter, 315
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, 258
implied object argument, 315
implicit conversion sequences, 315
non-static member function and, 315
import, 15, 443, 447, 471, 1630
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 directives,
source-file inclusion
inclusive-or-expression, 136, 1618
incomplete, 132
incompletely-defined object type, see object type,
incompletely-defined
increment operator
overloaded, see overloading, increment
operator
increment operator function, see operator
function, increment
incrementable, 917
independent_bits_engine
generation algorithm, 1168
state, 1167
textual representation, 1168
transition algorithm, 1167
indeterminate value, 60, see value, indeterminate
indeterminately sequenced, 76
indirect base class, see base class, indirect
indirection, 120
inheritance, 275
using-declaration and, 221
init-capture, 103, 1615
init-capture pack, 107, 370
init-declarator, 174, 1622
init-declarator-list, 174, 1622
init-statement, 146, 1619
initial suspend point, 208
initialization, 82, 188–204
aggregate, 192
array, 192
array of class objects, 196, 295
automatic, 153, 154
base class, 295, 296
by inherited constructor, 299
character array, 197
class member, 190
class object, see also constructor, 192,
294–301
cost, 166, 192
cost member, 297
constant, 82
costra and, 294
copy, 190
default, 189
default constructor and, 294
definition and, 157
direct, 190
dynamic, 82
dynamic block-scope, 154
dynamic non-local, 83
explicit, 294
jump past, 153
list-initialization, 199–204
local static, 154
local thread_local, 154
member, 295
member function call during, 299
member object, 296
order of, 83, 276
order of base class, 298
order of member, 298
order of virtual base class, 298
overloaded assignment and, 295
parameter, 111
reference, 180, 197
reference member, 297
static and thread, 82
static member, 269
union, 196, 274
vacuous, 57
virtual base class, 261
zero-initialization, 82, 189
initializer
base class, 204
member, 204
pack expansion, 299
scope of member, 298
temporary and declarator, 66
initializer, 188, 1623
initializer-clause, 189, 1623
initializer-list, 189, 1623
initializer-list constructor, 200
seed sequence requirement, 1157
<initializer_list>, 519, 1647
initializing declaration, 192
injected-class-name, 247
inline, 485
inline function, 165, see function, inline
inline namespace, see namespace, inline
inline namespace set, 215
inline variable, see variable, inline
instantiation explicit, 406
point of, 397
template implicit, 402
instantiation context, 244
instantiation units, 11
int, 16, 167, 1621
integer literal, see literal, integer
integer representation, 64
integer type, 71
integer-class type, see type, integer-class
integer-like, 917
integer-literal, 17, 1612
integer-suffix, 17, 1612
integral constant expression, see expression,
integral constant
integral promotion, 93
integral type, 71
implementation-defined sizeof, 70
inter-thread happens before, 78
interface dependency, 241
internal linkage, 51
interval boundaries
piecewise_constant_distribution, 1187
piecewise_linear_distribution, 1188
<inttypes.h>, 1485, 1663
invalid iterator, see iterator, invalid
invalid pointer value, see value, invalid pointer
invocation
macro, 450
<iomanip>, 494, 1383, 1402, 1404
<iost>, 1359
<iostfwd>, 1356, 1357
<iostream>, 388, 1358, 1365
istype
regular expression traits, 1488
<iso646.h>, 469, 1659, 1663, 1664
<stream>, 492, 1383, 1384
iteration-statement, 149, 152, 1619
iterator, 908
constexpr, 909
dereferenceable, 908
invalid, 909
past-the-end, 908
<iterator>, 492–494, 901, 957, 965, 1673
J
J_{n} (spherical Bessel functions), 1223
J_{\nu} (Bessell functions), 1220
Jessie, 267
jump-statement, 152, 1619
K
K (complete elliptic integrals), 1219
K_{\nu} (Bessell functions), 1220
key parameter, see parameter, key
keyword, 15, 1610
keyword, 15, 1611
L
L_{n} (Laguerre polynomials), 1222
L_{m}^{n} (associated Laguerre polynomials), 1218
label, 153
case, 147, 149
default, 147, 149
scope of, 35, 147
labeled-statement, 147, 1619
Laguerre polynomials
L_{n}, 1222
L_{m}^{n}, 1218
lambda-capture, 102, 1615
lambda-declarator, 98, 1615
lambda-expression, 98, 1615
lambda-introducer, 98, 167, 1615
language linkage, 226
<latch>, 493, 1592, 1638
lattice, see DAG, subobject
layout
bit-field, 270
class object, 253, 276
layout-compatible, 70
class, 254
enumeration, 212
layout-compatible type, 70
left shift
undefined, 133
left shift operator, see operator, left shift
left-pad, 734
Legendre functions Y_{m}^{n}, 1223
Legendre polynomials
P_{n}, 1222
P_{m}^{n}, 1219
letter, 466
lexical conventions, see conventions, lexical
LIA-1, 1684
library
C standard, 459, 466, 468, 471, 1658, 1659,
1663
C++ standard, 458, 483, 484, 487
library clauses, 8
lifetime, 57
limits
Index

implementation, 4
<limits>, 492, 494
<limits.h>, 501, 1663
line number, 454
line splicing, 10
linear_congruential_engine
generation algorithm, 1163
modulus, 1163
state, 1163
textual representation, 1163
transition algorithm, 1163
link, 1437
linkage, 26, 51–53
const and, 51
external, 51, 471, 482
implementation-defined object, 228
inline and, 51
internal, 51
module, 51
no, 51, 52
static and, 51
linkage specification, see specification, linkage
linkage-specification, 226, 1625
<list>, 492, 493, 818, 820, 957
list-initialization, 199
literal, 16–25, 95
base of integer, 17
boolean, 23
char16_t, 19
char32_t, 19
character, 18, 19
ordinary, 19
UTF-16, 19
UTF-32, 19
UTF-8, 19
wide, 19
complex, 1150
constant, 16
float, 20
floating-point, 20, 21
implementation-defined value of char, 19
integer, 17, 18
long, 17, 18
long double, 20
multicharacter, 19
implementation-defined value of, 19
narrow-character, 22
operator, 344
raw, 344
template, 344
template numeric, 344
template string, 344
pointer, 23
string, 21, 22
char16_t, 22
char32_t, 22
narrow, 22
raw, 12, 21
undefined change to, 23
UTF-16, 22
UTF-32, 22
UTF-8, 22
wide, 22
suffix identifier, 344
type of character, 19
type of floating-point, 20
type of integer, 18
unsigned, 17, 18
user-defined, 23
literal, 17, 1612
literal type, 69
literal-operator-id, 344, 1627
living dead
name of, 481
local class, see class, local
friend, 292
member function in, 254
scope of, 274
local entity, see entity, local
local scope, see block scope
local variable, 35
destruction of, 152, 153
local_iterator, 809
locale, 1486–1488, 1493
<locale>, 492, 1315, 1316, 1677
locale-specific behavior, see behavior, locale-specific
locale-specific form
format string, 726
<locale.h>, 1353, 1663
lock-free execution, 80
logical-and-expression, 136, 1618
logical-or-expression, 136, 1618
lognormal_distribution
probability density function, 1182
long, 16, 21, 167, 1621
typedef and, 158
long-long-suffix, 17, 1612
long-suffix, 17, 1612
lookup
argument-dependent, 42
class member, 45, 50
elaborated type specifier, 49
member name, 283
name, 26, 38–50
namespace aliases and, 50
namespace member, 46
qualified name, 44–49
template name, 385
unqualified name, 38
using-directives and, 50
lookup_classname
regular expression traits, 1488, 1519
lookup_collatename
regular expression traits, 1488
lower-order bit, 54
lowercase, 466
lparen, 442, 1630
lvalue, 87, 1653
lvalue reference, 179
Lvalue-Callable, 683

M
macro
active, 448
argument substitution, 450
definition, 448
function-like, 449, 450
arguments, 450
import, 447–449
masking, 485
name, 449
object-like, 449
point of definition, 448
point of import, 448
point of undefined, 448
pragma operator, 457
predefined, 455
replacement, 449–454
replacement list, 449
rescanning and replacement, 453
scope of definition, 454
main function, 82
implementation-defined linkage of, 82
implementation-defined parameters to, 82
parameters to, 82
return from, 82, 84
make progress
thread, 81
make-unsigned-like-t, 964
manifestly constant-evaluated, 144
<map>, 492, 493, 846, 847, 957
match_results
as sequence, 1503
matched, 1486
<math.h>, 1217, 1663
mathematical special functions, 1218–1223
max
random number distribution requirement, 1161
mean
normal_distribution, 1181
poisson_distribution, 1177
mem-initializer, 295, 1627
mem-initializer-id, 295, 1627
mem-initializer-list, 295, 1627
member
class static, 61
default initializer, 252
enumerator, 213
static, 251, 269
template and static, 367
member access operator
overloaded, 341
member candidate, 318
member data
static, 269
member function, 251
call undefined, 255
class, 254
const, 255
const volatile, 255
constexpr-compatible, 257
constructor and, 258
destructor and, 264
friend, 291
inline, 254
local class, 275
nested class, 294
non-static, 255
overload resolution and, 315
static, 251, 269
this, 256
union, 272
volatile, 255
member names, 35
member of an unknown specialization, 393
member of the current instantiation, see current instantiation, member of the
member pointer to, see pointer to member
member subobject, 55
member-declaration, 251, 1626
member-declarator, 251, 1626
member-declarator-list, 251, 1626
member-specification, 251, 1626
members, 35
<memory>, 492–494, 614, 615, 1134, 1674
memory location, 55
memory management, see new, delete
memory model, 54–55
<memory_resource>, 493, 653, 1642
mersenne_twister_engine
generation algorithm, 1164
state, 1164
textual representation, 1165
transition algorithm, 1164
message
diagnostic, 3, 7
min
random number distribution requirement, 1161
model
concept, 484
modifiable, 88
modification order, 77
module, 237
exported, 240
global, 237
named, 236
reserved name of, 236
module, 15, 442, 443, 447, 1629, 1630
module implementation unit, 236
module interface unit, 236
module partition, 236
module unit, 236

Index
module unit purview, see purview, module unit
module-declaration, 236, 1625
module-file, 442, 1629
module-import-declaration, 240, 1626
module-name, 236, 1625
module-name-qualifier, 236, 1625
module-partition, 236, 1625
more constrained, 363
more cv-qualified, 73
more specialized, 377, 424
class template, 377
function template, 424
most derived class, 56
most derived object, 56
bit-field, 56
zero size subobject, 56
move
class object, see constructor, move, see assignment operator, move
move, 564
multi-pass guarantee, 919, 923
multibyte character, see character, multibyte
multibyte encoding, see encoding, multibyte
multicharacter literal, see literal, multicharacter
multiple inheritance, 275, 276
virtual and, 280
multiple threads, see threads, multiple
mulitiplicative-expression, 132, 1617
mutable, 16, 158, 1620
mutable iterator, 908
<mutex>, 494, 1565, 1647
mutex types, 1566

N

n (spherical Neumann functions), 1223
N (Neumann functions), 1221
name, 15, 26, 53, 96
address of cv-qualified, 120
dependent, 391
elaborated
tenum, 169
exported, 239
global, 36
length of, 15
macro, see macro, name
point of declaration, see declaration, point of
predefined macro, see macro, predefined
qualified, 44
reserved, 481
same, 26
scope of, 32
unqualified, 38
zombie, 481
name class, see class name
name hiding, 33, 38, 97, 153
class definition, 249
function, 313
overloading versus, 313
user-defined conversion and, 266
using-declaration and, 224
name space
label, 147
named module, see module, named
named-namespace-definition, 214, 1624
namespace, 468
alias, 218
associated, 42
definition, 214
enclosing, 215
extend, 215
global, 15, 36
inline, 215
member definition, 216
unnamed, 216
namespace, 16, 214, 216, 218, 1624
namespace scope
global, 36
namespace-alias, 218, 1624
namespace-alias-definition, 218, 1624
namespace-body, 214, 1624
namespace-definition, 214, 1624
namespace-name, 214, 1624
namespaces, 214–220
NaN, 1218
narrow character type, 71
narrowing conversion, 203
native encoding, 1447
native pathname format, 1443
NDEBUG, 471
necessarily reachable, see reachable, necessarily needed
exception specification, 439
needed for constant evaluation, 145
negative_binomial_distribution
discrete probability function, 1177
nested class, see class, nested
local class, 275
scope of, 271
nested within, 56
nested-name-specifier, 97, 1615
nested-namespace-definition, 214, 1624
nested-requirement, 110, 1616
Neumann functions
N, 1221
n, 1223
<new>, 62, 493, 504
new, 16, 62, 124, 125, 339, 395, 1617
array of class objects and, 127
constructor and, 127
default constructor and, 127
exception and, 127
initialization and, 127
operator
replaceable, 483
scoping and, 125
storage allocation, 124

Index
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, 147
null wide character, see wide-character, null
nullptr, 16
number
hex, 19
octal, 19
preprocessing, 14
subnormal, 495, 496, 498, 499
<numbers>, 493, 1224, 1638
<numeric>, 493, 494, 1121
numeric type, see type, numeric
numeric_limits, 494
specializations for arithmetic types, 71

O
object, see also object model, 26, 55
byte copying and, 68
callable, 668
complete, 55
const, 73
cost volatile, 73
definition, 28
destructor and placement of, 265
destructor static, 84
exception, see exception handling, exception object
implicit creation, 57
linkage specification, 228
local static, 61
nested within, 56
nonzero size, 56
providing storage for, 55
reified, 965
suitable created, 57
unnamed, 258
volatile, 73
zero size, 56
object class, see class object
object expression, 113, 131
object lifetime, 57–60
object model, 55–57
object pointer type, 72
object temporary, see temporary object type, 69
incompletely-defined, 69
object-like macro, see macro, object-like
observable behavior, see behavior, observable
octal-digit, 17, 1612
octal-escape-sequence, 19, 1613
octal-literal, 17, 1612
odr-usable, 29
odr-use, 29

Index

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, 147
null wide character, see wide-character, null
nullptr, 16
number
hex, 19
octal, 19
preprocessing, 14
subnormal, 495, 496, 498, 499
<numbers>, 493, 1224, 1638
<numeric>, 493, 494, 1121
numeric type, see type, numeric
numeric_limits, 494
specializations for arithmetic types, 71

O
object, see also object model, 26, 55
byte copying and, 68
callable, 668
complete, 55
const, 73
cost volatile, 73
definition, 28
destructor and placement of, 265
destructor static, 84
exception, see exception handling, exception object
implicit creation, 57
linkage specification, 228
local static, 61
nested within, 56
nonzero size, 56
providing storage for, 55
reified, 965
suitable created, 57
unnamed, 258
volatile, 73
zero size, 56
object class, see class object
object expression, 113, 131
object lifetime, 57–60
object model, 55–57
object pointer type, 72
object temporary, see temporary object type, 69
incompletely-defined, 69
object-like macro, see macro, object-like
observable behavior, see behavior, observable
octal-digit, 17, 1612
octal-escape-sequence, 19, 1613
octal-literal, 17, 1612
odr-usable, 29
odr-use, 29

Index 1727
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-initialization</td>
<td>83</td>
</tr>
<tr>
<td>one-definition rule</td>
<td>28–32</td>
</tr>
<tr>
<td>opaque-enum-declaration</td>
<td>211, 1624</td>
</tr>
<tr>
<td>operating system dependent</td>
<td>1438</td>
</tr>
<tr>
<td>operator</td>
<td>16, 339</td>
</tr>
<tr>
<td>**=</td>
<td>139</td>
</tr>
<tr>
<td>+=</td>
<td>121, 139</td>
</tr>
<tr>
<td>-=</td>
<td>139</td>
</tr>
<tr>
<td>/=</td>
<td>139</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>139</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>139</td>
</tr>
<tr>
<td>%=</td>
<td>139</td>
</tr>
<tr>
<td>&amp;=</td>
<td>139</td>
</tr>
<tr>
<td>^=</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>addition</td>
<td>132</td>
</tr>
<tr>
<td>additive</td>
<td>132</td>
</tr>
<tr>
<td>address-of</td>
<td>120</td>
</tr>
<tr>
<td>assignment</td>
<td>139, 467</td>
</tr>
<tr>
<td>bitwise</td>
<td>135</td>
</tr>
<tr>
<td>bitwise AND</td>
<td>135</td>
</tr>
<tr>
<td>bitwise exclusive OR</td>
<td>136</td>
</tr>
<tr>
<td>bitwise inclusive OR</td>
<td>136</td>
</tr>
<tr>
<td>cast</td>
<td>120, 176</td>
</tr>
<tr>
<td>class member access</td>
<td>113</td>
</tr>
<tr>
<td>comma</td>
<td>140</td>
</tr>
<tr>
<td>comparison</td>
<td></td>
</tr>
<tr>
<td>constexpr-compatible</td>
<td>306</td>
</tr>
<tr>
<td>implicitly defined</td>
<td>306</td>
</tr>
<tr>
<td>secondary</td>
<td>308</td>
</tr>
<tr>
<td>conditional expression</td>
<td>136</td>
</tr>
<tr>
<td>copy assignment</td>
<td></td>
</tr>
<tr>
<td>copy</td>
<td></td>
</tr>
<tr>
<td>decrement</td>
<td>114, 120, 121</td>
</tr>
<tr>
<td>division</td>
<td>131</td>
</tr>
<tr>
<td>equality</td>
<td>134</td>
</tr>
<tr>
<td>defaulted</td>
<td>307</td>
</tr>
<tr>
<td>deleted</td>
<td>306</td>
</tr>
<tr>
<td>function call</td>
<td>111, 338</td>
</tr>
<tr>
<td>greater than</td>
<td>134</td>
</tr>
<tr>
<td>greater than or equal to</td>
<td>134</td>
</tr>
<tr>
<td>implementation</td>
<td>338</td>
</tr>
<tr>
<td>increment</td>
<td>114, 120, 121</td>
</tr>
<tr>
<td>indirect</td>
<td>120</td>
</tr>
<tr>
<td>inequality</td>
<td>134</td>
</tr>
<tr>
<td>defaulted</td>
<td>308</td>
</tr>
<tr>
<td>left shift</td>
<td>133</td>
</tr>
<tr>
<td>less than</td>
<td>134</td>
</tr>
<tr>
<td>less than or equal to</td>
<td>134</td>
</tr>
<tr>
<td>logical AND</td>
<td>136</td>
</tr>
<tr>
<td>logical negation</td>
<td>120, 121</td>
</tr>
<tr>
<td>logical OR</td>
<td>136</td>
</tr>
<tr>
<td>move assignment</td>
<td></td>
</tr>
<tr>
<td>move</td>
<td></td>
</tr>
<tr>
<td>multiplication</td>
<td>131</td>
</tr>
<tr>
<td>multiplicative</td>
<td>131</td>
</tr>
<tr>
<td>ones' complement</td>
<td>120, 121</td>
</tr>
<tr>
<td>overloaded</td>
<td>86, 338</td>
</tr>
<tr>
<td>pointer to member</td>
<td>131</td>
</tr>
<tr>
<td>pragma, see macro, pragma operator</td>
<td></td>
</tr>
<tr>
<td>precedence of</td>
<td>86</td>
</tr>
<tr>
<td>relational</td>
<td>134</td>
</tr>
<tr>
<td>defaulted</td>
<td>308</td>
</tr>
<tr>
<td>remainder</td>
<td>131</td>
</tr>
<tr>
<td>right shift</td>
<td>133</td>
</tr>
<tr>
<td>scope resolution</td>
<td>45, 97, 125, 254, 275, 281</td>
</tr>
<tr>
<td>side effects and comma</td>
<td>140</td>
</tr>
<tr>
<td>side effects and logical AND</td>
<td>136</td>
</tr>
<tr>
<td>side effects and logical OR</td>
<td>136</td>
</tr>
<tr>
<td>sizeof</td>
<td>120, 123</td>
</tr>
<tr>
<td>spaceship</td>
<td>133</td>
</tr>
<tr>
<td>subscribing</td>
<td>111, 338</td>
</tr>
<tr>
<td>subscripting</td>
<td>339</td>
</tr>
<tr>
<td>subtraction</td>
<td>132</td>
</tr>
<tr>
<td>three-way comparison</td>
<td>133</td>
</tr>
<tr>
<td>defaulted</td>
<td>307</td>
</tr>
<tr>
<td>deleted</td>
<td>306</td>
</tr>
<tr>
<td>unary</td>
<td>120</td>
</tr>
<tr>
<td>unary minus</td>
<td>120, 121</td>
</tr>
<tr>
<td>unary plus</td>
<td>120, 121</td>
</tr>
<tr>
<td>operator delete</td>
<td></td>
</tr>
<tr>
<td>destroying</td>
<td>63</td>
</tr>
<tr>
<td>operator delete, see also delete</td>
<td>125, 129, 308</td>
</tr>
<tr>
<td>operator function</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>339</td>
</tr>
<tr>
<td>class member access</td>
<td>341</td>
</tr>
<tr>
<td>comparison</td>
<td>339</td>
</tr>
<tr>
<td>decrement</td>
<td>341</td>
</tr>
<tr>
<td>equality</td>
<td>339</td>
</tr>
<tr>
<td>function call</td>
<td>340</td>
</tr>
<tr>
<td>increment</td>
<td>341</td>
</tr>
<tr>
<td>prefix unary</td>
<td>339</td>
</tr>
<tr>
<td>relational</td>
<td>339</td>
</tr>
<tr>
<td>simple assignment</td>
<td>339</td>
</tr>
<tr>
<td>subscripting</td>
<td>340</td>
</tr>
<tr>
<td>three-way comparison</td>
<td>339</td>
</tr>
<tr>
<td>operator new, see also new</td>
<td>125</td>
</tr>
<tr>
<td>operator overloading, see overloading, operator</td>
<td></td>
</tr>
<tr>
<td>operator use</td>
<td></td>
</tr>
<tr>
<td>scope resolution</td>
<td>269</td>
</tr>
<tr>
<td>operator!=</td>
<td></td>
</tr>
<tr>
<td>random number distribution requirement</td>
<td>1161</td>
</tr>
<tr>
<td>random number engine requirement</td>
<td>1159</td>
</tr>
<tr>
<td>operator()</td>
<td></td>
</tr>
<tr>
<td>random number distribution requirement</td>
<td>1161</td>
</tr>
<tr>
<td>random number engine requirement</td>
<td>1159</td>
</tr>
<tr>
<td>operator-or-punctuator</td>
<td>16, 1611</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td></td>
</tr>
<tr>
<td>random number distribution requirement</td>
<td>1162</td>
</tr>
<tr>
<td>random number engine requirement</td>
<td>1159</td>
</tr>
<tr>
<td>operator==</td>
<td></td>
</tr>
</tbody>
</table>
random number distribution requirement, 1161
random number engine requirement, 1159

operator>>
random number distribution requirement, 1162
random number engine requirement, 1159
operators
built-in, 86
optimization of temporary, see temporary,
elimination of
<optional>, 493, 581, 1642
optional object, 581
or, 16
or_eq, 16
order of evaluation in expression, see expression,
order of evaluation of
order of execution
base class constructor, 259
base class destructor, 264
constructor and array, 294
constructor and static data members, 295
destructor, 264
destructor and array, 265
member constructor, 259
member destructor, 264
ordering
function template partial, see template,
function, partial ordering
ordinary character literal, 19
ordinary string literal, 22
<ostream>, 492, 1383, 1394
over-aligned type, see type, over-aligned
overflow, 86
undefined, 86
overload resolution, 311
overload set, 38
overloaded function, see overloading
address of, 121, 337
overloaded operator, see overloading, operator
inheritance of, 339
overloading, 184, 249, 311–344, 378
access control and, 314
address of overloaded function, 337
argument lists, 315–325
array versus pointer, 312
assignment operator, 339
binary operator, 339
built-in operators and, 341
candidate functions, 315–325
declaration matching, 313
declarations, 311
element of, 311
function call operator, 340
function versus pointer, 312
member access operator, 341
operator, 338–341
prohibited, 311
resolution, 314–337

best viable function, 326–339
better viable function, 326
contexts, 314
function call syntax, 316–317
function template, 431
implicit conversions and, 328–337
initialization, 320, 321
operators, 318
scoping ambiguity, 284
template, 380
template name, 385
viable functions, 325–339
subscripting operator, 340
unary operator, 339
user-defined literal, 340
using directive and, 220
using-declaration and, 225
overloads
floating-point, 1150
override, 15, 251, 1626
overrider
final, 278
own, 630

P
P_\ell (Legendre polynomials), 1222
P_m^\ell (associated Legendre polynomials), 1219
pack, 370
unexpanded, 371
pack expansion, 370
pattern, 370
padding bits, 69
pair
tuple interface to, 567
parallel algorithm, 1026
parallel forward progress guarantees, 81
param
random number distribution requirement, 1161
seed sequence requirement, 1157

param_type
random number distribution requirement, 1161
parameter, 4
catch clause, 4
disqualifying, 542
function, 4
function-like macro, 4
key, 543
macro, 450
reference, 179
scope of, 35
template, 4, 27
void, 183
parameter declaration, 27
parameter list
variable, 112, 183
parameter mapping, 360
Index 1730

parameter-declaration, 183, 1623
parameter-declaration-clause, 182, 1623
parameter-type-list, 182, 1623
parameterized type, see template
parameters-and-qualifiers, 175, 1622
parent directory, 1437
past-the-end iterator, see iterator, past-the-end
path, 1443
path equality, 1457
pathname, 1443
pathname resolution, 1443
pattern, see pack expansion, pattern
perfect forwarding call wrapper, 669
period, 467
phase completion step, 1594
phase synchronization point, see barrier, phase synchronization point
phases of translation, see translation, phases
Π (complete elliptic integrals), 1220
Π (incomplete elliptic integrals), 1221
piecewise construction, 569
piecewise_constant_distribution
interval boundaries, 1187
probability density function, 1187
weights, 1187
piecewise_linear_distribution
interval boundaries, 1188
probability density function, 1188
weights at boundaries, 1188
placeholder type deduction, 172
placeholder-type-specifier, 170, 1622
placement new-expression, see new-expression, placement
plain lock-free atomic operation, 533
pm-expression, 131, 1617
POD, 1672
point, 72
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, 88
integer representation of safely-derived, 64
safely-derived, 63–64
strict total order, 460
to traceable object, 63, 487
zero, see value, null pointer
pointer literal, see literal, pointer
pointer past the end of, 72
pointer to, 72
pointer to member, 72, 131, 180
pointer-interconvertible, 72
pointer-literal, 23, 1614
Poisson distributions, 1177–1181
poisson_distribution
 discrete probability function, 1177
 mean, 1177
 polymorphic class, see class, polymorphic
 pool resource classes, 658
 pools, 658
 population, 1093
 POSIX, 2
 extended regular expressions, 1493
 regular expressions, 1493
 postfix ++ and --
 overloading, 341
 postfix ++, 114
 postfix --, 114
 postfix-expression, 110, 1616
 potential results, 28
 potential scope, 33
 potentially concurrent, 79
 potentially constant evaluated, 145
 potentially evaluated, 28
 potentially-constant, 140
 potentially-overlapping subobject, 56
 potentially-throwing
 exception specification, 437
 expression, 438
 pp-global-module-fragment, 442, 1629
 pp-import, 447, 1630
 pp-module, 447, 1630
 pp-number, 14, 1611
 pp-private-module-fragment, 442, 1629
 pp-tokens, 443, 1630
 precedence of operator, see operator, precedence of
 preferred-separator, 1446
 prefix
 L, 19, 22
 R, 21
 U, 19, 22
 u, 19, 22
 u8, 19, 22
 prefix ++ and --
 overloading, 341
 prefix ++, 121
 prefix --, 121
 prefix unary operator function, see operator function, prefix unary
 preprocessing, 444
 preprocessing directive, 442–457
 conditional inclusion, 444
 error, 454
 header inclusion, 446
 import, 447
 line control, 454
 macro replacement, see macro, replacement
 module, 447
 null, 455
 pragma, 454
source-file inclusion, 446
preprocessing-file, 442, 1629
preprocessing-op-or-punc, 16, 1611
preprocessing-operator, 16, 1611
preprocessing-token, 12, 1610
primary class template, see template, primary
class, 1486
primary module interface unit, 236
primary-expression, 95, 1615
private, 16, 243, 275, see access control,
private, 442, 1626, 1627, 1629
private-module-fragment, 243, 1626
probability density function
cauchy_distribution, 1183
chi_squared_distribution, 1183
exponential_distribution, 1178
extreme_value_distribution, 1180
fisher_f_distribution, 1184
gamma_distribution, 1179
lognormal_distribution, 1182
normal_distribution, 1181
piecewise_constant_distribution, 1187
piecewise_linear_distribution, 1188
student_t_distribution, 1185
uniform_real_distribution, 1174
weibull_distribution, 1180
program, 51
ill-formed, 4
startup, 82–84
termination, 84–85
well-formed, 5, 8
program execution, 7–76
abstract machine, 7
as-if rule, see as-if rule
program semantics
affected by the existence of a variable or
function definition, 404
projection, 461
promise object, 208
promise type, see coroutine, promise type
promoted integral type, 341
promotion
bool to int, 93
default argument promotion, 113
floating-point, 93
integral, 92
prospective destructor, see destructor, prospective
protected, 16, 275, see access control,
protected, 1627
protection, see access control, 486
prototype parameter
concept, 384
provides storage, 55
prvalue, 87
pseudo-destructor, 98
ptr_abstract_declarator, 176, 1622
ptr_declarator, 175, 1622
ptr_operator, 175, 1622
ptrdiff_t, 132
implementation-defined type of, 132
public, 16, 275, see access control, public, 1627
punctuator, 16
pure-specifier, 251, 1626
purview
module, 237
module unit, 237
named module, 237
Q
q-char, 14, 1611
q-char-sequence, 14, 1611
qualification
explicit, 44
qualified-id, 97, 1615
qualified-namespace-specifier, 218, 1624
<queue>, 886, 887
R
r-char, 21, 1614
r-char-sequence, 21, 1614
<random>, 1154, 1647
random number distribution
bernoulli_distribution, 1175inomial_distribution, 1175
cauchy_distribution, 1183
chi_squared_distribution, 1183
discrete_distribution, 1185
exponential_distribution, 1178
extreme_value_distribution, 1180
fisher_f_distribution, 1184
gamma_distribution, 1179
lognormal_distribution, 1182
normal_distribution, 1181
piecewise_constant_distribution, 1187
piecewise_linear_distribution, 1188
student_t_distribution, 1185
uniform_real_distribution, 1174
weibull_distribution, 1180
random number distributions
Bernoulli, 1175–1177
normal, 1181–1185
Poisson, 1177–1181
sampling, 1185–1190
uniform, 1173–1175
random number engine
linear_congruential_engine, 1163
ersennete_twister_engine, 1164
requirements, 1158–1159
subtract_with_carry_engine, 1165
with predefined parameters, 1169–1170
random number engine adaptor
discard_block_engine, 1166
independent_bits_engine, 1167
shuffle_order_engine, 1168
with predefined parameters, 1169–1170
random number generation, 1153–1190
distributions, 1173–1190
engines, 1162–1169
predefined engines and adaptors, 1169–1170
requirements, 1156–1162
synopsis, 1154–1156
utilities, 1171–1173
random number generator, see uniform random
bit generator
random_device
implementation leeway, 1170
range, 909
counted, 909, 1014
<ranges>, 494, 579, 960, 965, 1638, 1672, 1673
<ratio>, 713, 1647
raw string literal, 21
raw-string, 21, 1614
reachable
declaration, 245
necessarily
translation unit, 245
translation unit, 245
reachable from, 909
ready, 1503, 1598
redefinition
typedef, 161
ref-qualifier, 175, 1622
reference, 72
assignment to, 139
call by, 112
forwarding, 420
lvalue, 72
null, 180
rvalue, 72
sizeof, 123
reference collapsing, 180
reference lifetime, 57
reference-compatible, 197
reference-related, 197
references
normative, 2
<regex>, 493, 957, 1488, 1647
regex_iterator
end-of-sequence, 1513
regex_token_iterator
end-of-sequence, 1514
regex_traits
specializations, 1496
region
declarative, 26, 32
intervening, 33
register, 15, 16
register storage class, 1641
regular expression, 1486–1519
grammar, 1517
matched, 1486
requirements, 1487
regular expression traits, 1517
char_class_type, 1487
isctype, 1488
lookup_classname, 1488, 1519
lookup_collatename, 1488
requirements, 1487, 1496
transform, 1487, 1518
transform_primary, 1488, 1519
translate, 1487, 1518
translate_nocase, 1487, 1518
reified object, see object, reified
reinterpret_cast, 16, see cast, reinterpret, 110,
395, 396, 1616
relational operator function, see operator function,
relational
relational-expression, 134, 1618
relative path, 1443
relative-path, 1445
relaxed pointer safety, 64
release sequence, 77
remainder operator, see operator, remainder
remote time zone database, 1295
replacement
macro, see macro, replacement
replacement field
format string, 723
replacement-list, 443, 1630
representation
object, 68
value, 68
represents the address, 72
requirement, 108
compound, 109
nested, 110
simple, 108
type, 109
requirement, 108, 1616
requirement-body, 107, 1616
requirement-parameter-list, 107, 1616
requirement-seq, 108, 1616
requirements, 463
container, 785, 808, 821, 822, 1503
not required for unordered associated containers, 807
iterator, 908
numeric type, 1141
random number distribution, 1160–1162
random number engine, 1158–1159
regular expression traits, 1487, 1496
seed sequence, 1156–1157
sequence, 1503
uniform random bit generator, 1157
unordered associative container, 808
requires, 16, 107, 109, 110, 346, 1616, 1628
requires-clause, 346, 1628
trailing, 175
requires-expression, 107, 1616

Index
rescanning and replacement, see macro, rescanning and replacement
reserved identifier, 15
reset, 630
reset
  random number distribution requirement, 1161
resolution, see overloading, resolution restriction, 484, 485, 487, 1664
  address of bit-field, 270
  anonymous union, 274
  bit-field, 270
  constructor, 258
  destructor, 264
 extern, 159
 local class, 275
 operator overloading, 338
 overloading, 339
 pointer to bit-field, 270
 reference, 180
 static, 159
 static member local class, 275
 union, 272
result
glvalue, 88
prvalue, 88
result object, 88
result_type
  entity characterization based on, 1153
  random number distribution requirement, 1161
  seed sequence requirement, 1157
rethrow, see exception handling, rethrow
return, 16, 152, 1619
  and handler, 433
  and try block, 433
  constructor and, 152
  reference and, 197
return statement, see return
return type, 184
covariant, 279
overloading and, 311
return-type-requirement, 109, 1616
rewritten candidate, 318
right shift operator, see operator, right shift
root-directory, 1445
root-name, 1445
rounding, 93
rvalue, 87
  lvalue conversion to, see conversion,
  lvalue-to-rvalue, 1653
rvalue reference, 179
S
s-char, 21, 1614
s-char-sequence, 21, 1613
safely-derived pointer, 64
  integer representation, 64
sample, 1093
sampling distributions, 1185–1190
satisfy, see constraint, satisfaction
scalar type, see type, scalar
scope, 1, 26, 32–38, 157
  anonymous union at namespace, 274
  block, 35
  class, 36
  declarations and, 32–34
  destructor and exit from, 152
  enumeration, 37
  exception declaration, 35
  function, 35
  function parameter, 35
  function prototype, see scope, function
    parameter
  global, 36
  iteration-statement, 149
  macro definition, see macro, scope of
definition
  name lookup and, 38–50
  namespace, 35
  overloading and, 313
  potential, 32
  selection-statement, 147
  template parameter, 37
scope name hiding and, 38
scope resolution operator, see operator, scope resolution
scoped enumeration, see enumeration, scoped
<scoped_allocator>, 492, 662, 1647
secondary comparison operator, 308
seed
  random number engine requirement, 1158
  seed sequence, 1156
  requirements, 1156–1157
  selected destructor, see destructor, selected
  selection-statement, 147, 1619
semantics
  class member, 113
<semaphore>, 494, 1591, 1638
sentinel, 909
separate compilation, see compilation, separate
separate translation, see compilation, separate
sequence constructor
  seed sequence requirement, 1157
sequenced after, 76
sequenced before, 76
sequencing operator, see operator, comma
<set>, 492, 493, 846, 848, 957
<setjmp.h>, 532, 1663
shared lock, 1570
shared mutex types, 1570
shared state, see future, shared state
shared timed mutex type, 1572
<shared_mutex>, 494, 1566, 1645
shift operator
  left, see operator, left shift
  right, see operator, right shift
program, 471, 483
state, 605
discard_block_engine, 1166
independent_bits_engine, 1167
linear_congruential_engine, 1163
mersenne_twister_engine, 1164
shuffle_order_engine, 1168
subtract_with_carry_engine, 1165
state entity, 668
statement, 146–155
continue in for, 151
break, 152
compound, 147
continue, 152
declaration, 153
declaration in for, 151
declaration in if, 146
declaration in switch, 146, 149
declaration in while, 150
do, 149, 150
empty, 147
expression, 147
callthrough, 232
for, 149, 150
goto, 147, 152, 153
if, 147, 148
iteration, 149–152
jump, 152
labeled, 147
null, 147
range based for, 151
selection, 147–149
switch, 147, 149, 152
while, 149, 150
statement, 146, 1619
statement-seq, 147, 1619
static, 16, 158, 1620
destruction of local, 154
linkage of, 51, 159
overloading and, 311
static data member, 251
static initialization, 82
static member, 251
static member function, 251
static storage duration, 61
static type, see type, static
static_assert, 157
static_assert, 16, 156, 1620
not macro, 550
static_assert-declaration, 156, 1620
static_cast, 16, see cast, static, 110, 395, 396, 1616
STATICALLY-WIDEN, 1225
<string>, 492–494, 741, 744, 957
string literal, see literal, string
string_literal, 21, 1613
<strbuf>, 781, 1663
<string_view>, 492–494, 770, 957, 1642
stringize, see # operator
stringizing argument, 452
strongly happens before, 78
<stream>, 1664
struct
standard-layout, 249
struct, 16, 211, 247, 1624, 1626
structural type, see type, structural
structure tag, see class name
structured binding, 210
structured binding declaration, 157, 210
student_t_distribution
probability density function, 1185
sub-expression
regular expression, 1487
subexpression, 75
subnormal number, see number, subnormal
subobject, see also object model, 55
subscripting operator
overloaded, 340
<stdalign.h>, 1659, 1663, 1664
<stddarg.h>, 532, 1663
<static.h>
absence thereof, 468, 1659
<stdbool.h>, 1653, 1659, 1663, 1664
<stddef.h>, 19, 22, 489, 490, 1660, 1663
<stdexcept>, 547
<stdint.h>, 503, 1485, 1663
<stdio.h>, 1483, 1663
<stdlib.h>, 490, 1663, 1664
<stdnoreturn.h>
absence thereof, 468, 1659
stop request, 1553
stop state, 1553
<stop_token>, 493, 1554, 1638
storage class, 26
storage duration, 61–64
automatic, 61
class member, 64
dynamic, 61–64, 125
local object, 61
static, 61
thread, 61
storage management, see new, delete
storage-class-specifier, 158, 1620
stream
arbitrary-positional, 459
repositional, 461
<streambuf>, 1375
strict pointer safety, 64
<streambuf>, 1664
string
distinct, 23
null terminator, 747
null-terminated byte, see NTBS
null-terminated character type, 460
null-terminated multibyte, see NTMBS
sizeof, 23
type of, 22
width, 726
<string>, 492–494, 741, 744, 957
string literal, see literal, string
<strstream>, 1664
struct
standard-layout, 249
<stlalign.h>, 1659, 1663, 1664
<stlalign.h>, 1659, 1663, 1664
<stlalign.h>, 19, 22, 489, 490, 1660, 1663
<stlalign.h>, 1659, 1663, 1664
<stlalign.h>, 1659, 1663, 1664
<stlalign.h>, 19, 22, 489, 490, 1660, 1663
subscripting operator function, see operator function, subscripting

subsequence rule
  overloading, 335
substatement, 146
substitutability, 521
subsume, see constraint, subsumption

subtract_with_carry_engine
  carry, 1165
  generation algorithm, 1165
  state, 1165
  textual representation, 1166
  transition algorithm, 1165

subtraction
  implementation-defined pointer, 132
subtraction operator, see operator, subtraction

suffix
  F, 20
  f, 20
  L, 18, 20
  l, 18, 20
  U, 18
  u, 18

suitable created object, see object, suitable created

summary
  compatibility with ISO C, 1650
  compatibility with ISO C++ 2003, 1645
  compatibility with ISO C++ 2011, 1643
  compatibility with ISO C++ 2014, 1640
  compatibility with ISO C++ 2017, 1633
  syntax, 1610

surrogate call function, 317
swappable, 473
swappable with, 471

switch, 16, 147, 149, 1619
  and handler, 433
  and try block, 433
symbolic link, 1438
synchronize with, 77
<syncstream>, 494, 1433, 1638

synonym, 218
  type name as, 160
syntax
  class member, 113
synthesized three-way comparison, see three-way comparison, synthesized
<system_error>, 552, 554, 1647

T
target object, 668

template, 346–432
  alias, 383
  class, 365
  deducible, 167
  deducible arguments of, 323
  function, 413
  abbreviated, 185
equivalent, see equivalent, function template
  functionally equivalent, see functionally equivalent, function template
  key parameter of, 543
  partial ordering, 380
  member function, 366
  primary, 374
  static data member, 346
  variable, 346

template instantiation, 401
template name
  linkage of, 347
template parameter, 27
template parameter object, 349
template parameter pack, 369
template parameter scope, 37
template-argument, 351, 1628
default, 349
template-argument-equivalent, 364
template-argument-list, 351, 1628
template-declaration, 346, 1627
template-head, 346, 1627
template-id, 351, 1628
  valid, 352
template-name, 351, 1628
template-parameter, 347, 1628

template-parameter-list, 346, 1628
templated, 347
temporary, 65
  constructor for, 66
  destruction of, 66
  destructor for, 66
  elimination of, 65, 303
  implementation-defined generation of, 65
  order of destruction of, 66
terminate, 440
  called, 139, 435, 438, 440
termination
  program, 82, 85
terminology
  pointer, 72
text-line, 442, 1630
textual representation
  discard_block_engine, 1167
  independent_bits_engine, 1168
  shuffle_order_engine, 1169
  subtract_with_carry_engine, 1166
<tgmath.h>, 1659, 1663, 1664
this, 16, 95, 102, 256, 1615
type of, 256
this pointer, see this
thread, 77
<thread>, 493, 1558, 1647
thread of execution, 77
thread storage duration, see storage duration, thread
thread_local, 16, 158, 406, 409, 1620
threads
multiple, 77–81
<threads.h>
absence thereof, 468, 1659
three-way comparison
synthesized, 307
three-way comparison operator function, see
operator function, three-way comparison
throw, 16, 138, 395, 1618
throw-expression, 138, 1618
throwing, see exception handling, throwing
<time.h>, 1313, 1663
timed mutex types, 1568
to-unsigned-like, 964
token, 13
alternative, 13
directive-introducing, 443
preprocessing, 12–13
token, 13, 1610
traceable pointer object, 63, 487
trailing requires-clause, see requires-clause, trailing
trailing-return-type, 175, 1622
traits, 462
transform
regular expression traits, 1487, 1518
transform_primary
regular expression traits, 1488, 1518, 1519
transition algorithm
discard_block_engine, 1167
independent_bits_engine, 1167
linear_congruential_engine, 1163
mersenne_twister_engine, 1164
shuffle_order_engine, 1168
subtract_with_carry_engine, 1165
translate
regular expression traits, 1487, 1518
translate_nocase
regular expression traits, 1487, 1518
translation
phases, 10–11
separate, see compilation, separate
translation unit, 10, 51
name and, 26
translation-unit, 51, 1614
transparently replaceable, 59
trigraph sequence, 1640
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, 16
truncation, 93
try, 16, 433
try block, see exception handling, try block
try-block, 433, 1629
TU-local
entity, 53
value or object, 53
<tuple>, 492–494, 563, 571, 579, 1647, 1672, 1673
tuple
and pair, 567
type, 26, 68–73
allocated, 124
arithmetic, 71
promoted, 341
array, 72
bitmask, 466
Boolean, 71
callable, 668
char, 71
char16_t, 19, 22, 71, 74
char32_t, 19, 22, 71, 74
char8_t, 71
character, 71
class and, 247
compound, 72
cost, 165
cv-combined, 92
cv-unqualified, 73
destination, 191
double, 71
dynamic, 3, 4
enumerated, 72, 465, 466
example of incomplete, 69
extended integer, 71
extended signed integer, 70
extended unsigned integer, 70
float, 71
floating-point, 71
function, 72, 182, 183
fundamental, 71
implementation-defined sizeof, 70
implicit-lifetime, 69
incomplete, 28, 30, 34, 69, 90, 111, 113, 114,
120, 123, 124, 129, 275
incompletely-defined object, 69
int, 70
integral-class, 916
integral, 71
promoted, 341
long, 70
long double, 71
long long, 70
ordinary character, 71
over-aligned, 65
pointer, 72
polymorphic, 278
program-defined, 461
referenceable, 461
scalar, 69
short, 70
signed char, 70, 71
signed integer, 70

Index
signed-integer-class, 916
similar, see similar types
standard integer, 71
standard signed integer, 70
standard unsigned integer, 70
standard-layout, 69
static, 5
structural, 349
trivial, 69
trivially copyable, 68, 69
underlying
    char16_t, 71, 92
    char32_t, 71, 92
    char8_t, 71
enumeration, 92, 212
fixed, 212
wchar_t, 71, 92
unsigned, 70
unsigned char, 70, 71
unsigned int, 70
unsigned integer, 70
unsigned long, 70
unsigned long long, 70
unsigned short, 70
unsigned-integer-class, 916
void, 71
volatile, 165
wchar_t, 19, 22, 71, 74
type checking
    argument, 112
type concept, see concept, type
type conversion, explicit, see casting
type generator, see template
type name, 176
    nested, 272
    scope of, 272
type pun, 119
type specifier
    auto, 170
    bool, 167
    char, 167
    char16_t, 167
    char32_t, 167
    char8_t, 167
    const, 166
dec_type, 169
dec_type(auto), 170
double, 167
    elaborated, 49, 168
enum, 168
    float, 167
    int, 167
    long, 167
    short, 167
    signed, 167
    simple, 167
    unsigned, 167
    void, 167
    volatile, 166, 167
wchar_t, 167
type-constraint, 348, 1628
type-id, 176, 1622
type-id-only context, 386
type-name, 167, 1621
type-parameter, 347, 1628
type-parameter-key, 348, 1628
type-requirement, 109, 1616
type-specifier, 165, 1621
type-specifier-seq, 165, 1621
type_info, 115
[type_traits>, 492–494, 689, 1647, 1672
typedef
    function, 184
typedef, 16, 158, 1620
    overloading and, 312
typedef-name, 160, 1621
typeid, 16, 110, 115, 395, 396, 1616
    construction and, 303
    destruction and, 303
[typeindex>, 715, 1647
[typeinfo>, 116, 511
typename, 16, 109, 167, 168, 220, 322, 348, 385,
    1616, 1624, 1628
typename-specifier, 385, 1628
types
    implementation-defined, 465
U
<uchar.h>, 783, 1663
ud-suffix, 24, 1614
unary fold, 107
unary left fold, 107
unary operator
    interpretation of, 339
    overloaded, 339
unary right fold, 107
unary-expression, 120, 1617
unary-operator, 120, 1617
unblock, 5
undefined, 461, 462, 481, 482, 484, 1201, 1205,
    1208, 1368
undefined behavior, see behavior, undefined
underlying type, 71, see type, underlying
unevaluated operand, 89
Unicode required set, 457
uniform distributions, 1173–1175
uniform random bit generator
    requirements, 1157
uniform_int_distribution
    discrete probability function, 1173
uniform_real_distribution
    probability density function, 1174
union, 272
    standard-layout, 249
union, 16, 72, 247, 272, 273, 1626
anonymous, 273
    global anonymous, 274
union-like class, 274
unique pointer, 630
unit
   translation, 471, 482
universal character name, 10
universal-character-name, 11, 1610
Unix time, 1251
unnamed bit-field, 270
unnamed-namespace-definition, 214, 1624
unordered associative containers, 808
   complexity, 807
   equality function, 807
   equivalent keys, 807, 808, 873, 882
   exception safety, 818
   hash function, 807
   iterator invalidation, 818
   iterators, 818
   lack of comparison functions, 807
   requirements, 807, 808, 818
   unique keys, 807, 808, 866, 877
<unordered_map>, 492–494, 864, 957, 1647
unordered_map
   element access, 871
   unique keys, 806
unordered_multimap
   equivalent keys, 873
unordered_multiset
   equivalent keys, 882
<unordered_set>, 492, 493, 864, 865, 957, 1647
unordered_set
   unique keys, 877
unqualified-id, 97, 1615
unscoped enumeration, see enumeration, unscoped
unsequenced, 76
unsigned, 16, 167, 1621
   typedef and, 158
unsigned integer type, 70
unsigned-integer-class type, see type,
   unsigned-integer-class
unsigned-integer-like, 917
unsigned-suffix, 17, 1612
unspecified, 506, 507, 512, 1097, 1412, 1666, 1668
unspecified behavior, see behavior, unspecified
unwinding
   stack, 435
uppercase, 15, 466
upstream, 660
upstream allocator, 658
usable
   binary operator expression, 306
   usable candidate, see candidate, usable
   usable in constant expressions, 140
   user-defined conversion sequence, see conversion
   sequence, user-defined
user-defined literal, see literal, user-defined
   overloaded, 344
user-defined-character-literal, 23, 1614
user-defined-floating-point-literal, 23, 1614
user-defined-integer-literal, 23, 1614
user-defined-literal, 23, 1614
user-defined-string-literal, 23, 1614
user-provided, 206
uses-allocator construction, 625
using, 16, 156, 216, 218, 220, 229, 1620, 1624,
   1625
using-declaration, 220–226
using-declaration, 220, 1624
using-declarator, 220, 1624
using-declarator-list, 220, 1624
using-directive, 218–220
using-directive, 218, 1624
using-enum-declaration, 213, 1624
usual arithmetic conversions, see conversion, usual
   arithmetic
usual deallocation function, 63
UTF-16 character literal, 19
UTF-16 string literal, 22
UTF-32 character literal, 19
UTF-32 string literal, 22
UTF-8 character literal, 19
UTF-8 string literal, 22
<utility>, 473, 492–494, 561, 579, 1647, 1664,
   1672, 1673
V
va-opt-replacement, 450, 1630
vacuous initialization, 57
<vector>, 1190, 1192
valid but unspecified state, 462
value, 69
call by, 112
denormalized, see number, subnormal
indeterminate, 60
invalid pointer, 72
null member pointer, 94
null pointer, 72, 93
undefined unrepresentable integral, 93
value category, 87
value computation, 66, 75–76, 79, 114, 127, 139
value type, 908
value-initialization, 190
variable, 26
   function-local predefined, 204
   indeterminate uninitialized, 189
   inline, 165
   needed for constant evaluation, 145
   program semantics affected by the existence
   of a variable definition, 404
   variable arguments, 450
   variable template
      definition of, 346
<vector>, 494, 593, 1642, 1673
variant member, 274
<vector>, 492, 493, 818, 821, 957
vectorization-unsafe, 1026
<vector>, 492, 1638
virt-specifier, 251, 1626
Index 1740
Index of grammar productions

The first bold page number for each entry 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.

abstract-declarator, 176, 176, 183, 185, 186, 1622
abstract-pack-declarator, 176, 1623
access-specifier, 275, 275, 287, 288, 1627
additive-expression, 132, 1617
alias-declaration, 27, 52, 156, 160, 161, 183, 238,
242, 251, 346, 365, 383, 1620
alignment-specifier, 229, 230, 370, 1625
and-expression, 136, 1618
asm-declaration, 142, 156, 226, 226, 1625
assignment-expression, 139, 157, 171, 173, 186,
193, 196, 210, 270, 294, 320, 1618
assignment-operator, 139, 1618
attribute, 229, 229, 230, 370, 1625
attribute-argument-clause, 229, 229, 230–235,
1625
attribute-declaration, 27, 156, 157, 1620
attribute-list, 229, 229, 231–235, 370, 1625
attribute-namespace, 229, 229, 230, 1625
attribute-scoped-token, 229, 229, 230, 1625
attribute-specifier, 229, 229, 230, 1625
attribute-specifier-seq, 100, 125, 146, 147, 149, 157,
158, 160, 166, 169, 177–183, 204, 211,
212, 215, 216, 218, 226, 229, 229, 230,
233, 236, 240, 247, 252, 257, 263, 270,
275, 406, 433, 1625
attribute-token, 15, 229, 229, 230–235, 445, 482,
1625
attribute-using-prefix, 229, 229, 1625
await-expression, 121, 121, 122, 142, 207, 208, 532,
1617
balanced-token, 229, 1625
balanced-token-seq, 229, 229, 230, 1625
base-clause, 247, 275, 1627
base-specifier, 39, 45, 275, 275, 276, 283, 287, 290,
370, 1627
base-specifier-list, 263, 275, 275, 287, 298, 299,
306, 370, 371, 1627
binary-digit, 17, 17, 1612
binary-exponent-part, 20, 21, 1613
binary-literal, 17, 18, 1612
block-declaration, 156, 1620
boolean-literal, 23, 1614
brace-or-equal-initializer, 74, 189, 189, 190, 193,
252, 258, 270, 395, 1623
braced-list, 65, 74, 111, 113, 125, 139, 145, 153,
173, 189, 191, 194, 199, 200, 202, 295,
296, 298, 322, 391, 396, 412, 1623, 1632,
1641
c-char, 18, 19, 22, 1613
c-char-sequence, 11, 18, 1613
capture, 102, 370, 1615
capture-default, 30, 102, 103–105, 1615, 1634,
1661
capture-list, 102, 370, 1615
cast-expression, 107, 122, 129, 130, 130, 131, 145,
309, 339, 370, 372, 396, 1617
character-literal, 10, 11, 18, 19, 20, 23, 445, 449,
452, 1325, 1612, 1644, 1650
class-head, 33, 230, 247, 247, 1626
class-head-name, 216, 247, 247, 352, 1626
class-key, 49, 157, 169, 216, 247, 247, 249, 250,
272, 288, 366, 1626
class-name, 9, 34, 38, 49, 50, 157, 161, 169, 247,
247, 250, 264, 1626
class-or-decltype, 247, 275, 275, 296, 352, 385,
1627
class-specifier, 33, 53, 157, 166, 247, 247, 252, 256,
1626
class-virt-specifier, 247, 247, 698, 1626
compare-expression, 133, 1618
compound-requirement, 109, 109, 110, 1616
compound-statement, 75, 82, 102, 103, 105–107,
121, 146, 147, 147, 149, 153, 258, 297,
298, 305, 405, 433, 434, 436, 437, 1619
concept-definition, 27, 384, 384, 1628
concept-name, 53, 241, 353, 384, 1628
condition, 35, 146, 146, 147–149, 151, 153, 190,
191, 433, 1619
conditional-expression, 136, 137, 459, 1618
conditionally-supported-directive, 442, 443, 450,
1630
constant-expression, 3, 41, 75, 125, 140, 140, 144,
149, 157, 160, 181, 211, 212, 230, 252,
270, 315, 438, 464, 1618
constraint-expression, 96, 110, 175, 313, 347, 348,
353, 359, 361, 362, 380, 384, 464,
1628
constraint-logical-and-expression, 346, 1628
constraint-logical-or-expression, 175, 346, 347,
1628
coroutine-line, 442, 448, 1629
coroutine-declarator, 268, 1627
coroutine-function-id, 26, 45, 50, 97, 98, 170, 267,
268, 347, 395, 1627, 1636
coroutine-type-id, 45, 50, 98, 267, 268, 317, 340,
1627
coroutine-return-statement, 153, 207, 1619
Index of grammar productions

- nested-name-scoped-definition, 214, 215, 1624
- nested-requirement, 109, 110, 110, 1616
- new-declarator, 124, 124, 1617
- new-initializer, 67, 124, 124, 125, 127, 191, 259, 1617
- new-line, 443, 448, 1630
- new-placement, 124, 126, 128, 1617
- new-type-id, 73, 124, 124, 125, 171, 174, 386, 395, 1617
- nodeclspec-function-declaration, 156, 156, 157, 1620
- noexcept-expression, 123, 124, 1617
- noexcept-specification, 208
- nondigit, 14, 447, 1611
- nonzero-digit, 17, 1612
- noptptr-abstract-declarator, 176, 1622
- noptptr-abstract-pack-declarator, 176, 1623
- noptptr-declarator, 175, 1622
- noptptr-new-declarator, 124, 125, 1617
- octal-digit, 17, 17, 1612
- octal-escape-sequence, 19, 1613
- octal-literal, 17, 18, 1612
- opaque-enum-declaration, 27, 34, 211, 211, 212, 216, 252, 1624
- operator, 338, 1627
- operator-or-punctuator, 16, 16, 1611
- parameter-declaration-list, 182, 420, 425, 1623
- parameters-and-qualifiers, 175, 1622
- pathname, 1445
- placeholder-type-specifier, 167, 168, 170, 170, 173, 185, 1622
- pm-expression, 131, 131, 1617
- pointer-literal, 23, 1614
- pp-global-module-fragment, 442, 443, 1629
- pp-import, 443, 447, 1630
- pp-module, 447, 447, 1630
- pp-number, 14, 444, 445, 1611, 1640, 1643
- pp-private-module-fragment, 442, 1629
- pp-tokens, 443, 444, 445, 448, 451, 457, 1630
- preferred-separator, 1445, 1446, 1446, 1450–1452
- preprocessing-file, 442, 1629
- preprocessing-op-or-punc, 16, 1611
- preprocessing-operator, 16, 1611
- preprocessing-token, 12, 13, 444, 1610, 1630
- primary-expression, 95, 96, 317, 1615
- private-module-fragment, 30, 54, 171, 237, 240, 243, 243, 244, 259, 1626
- ptr-abstract-declarator, 176, 1622
- ptr-declarator, 175, 257, 263, 1622
- ptr-operator, 175, 1622
- pure-specifier, 251, 252, 282, 1626
- q-char, 14, 1611
- q-char-sequence, 14, 14, 1611
- qualified-name-specifier, 218, 218, 1624
- r-char, 12, 21, 1614
- r-char-sequence, 11, 21, 1614
- raw-string, 21, 21, 1614
- ref-qualifier, 5, 116, 131, 175, 179, 180, 182–184, 205, 210, 224, 256, 257, 306, 311, 315, 335, 342, 381, 461, 1622
- relational-expression, 134, 1618
- relative-path, 1445, 1446, 1455
- replacement-list, 443, 449, 1630
- requirement, 108, 108, 109, 1616
- requirement-body, 107, 108, 1616
- requirement-parameter-list, 107, 108, 386, 1616
- requirement-seq, 108, 1616
- requires-expression, 107, 107, 108, 109, 534, 1633
- return-type requirement, 109, 109, 1616
- root-directory, 1443, 1445, 1446, 1453, 1456
- root-name, 1443, 1445, 1446, 1450, 1453, 1455, 1456
- s-char, 21, 1614
- s-char-sequence, 11, 21, 1613
- selection-statement, 146, 147, 147, 1619
- shift-expression, 133, 1617
- sign, 20, 21, 1613, 1640
- simple-capture, 30, 102, 103, 104, 107, 1615
- simple-declaration, 121, 156, 156, 157, 174, 386, 407, 1620
- simple-escape-sequence, 19, 1613
- simple-requirements, 108, 109, 110, 1616
<table>
<thead>
<tr>
<th>Index of grammar productions 1745</th>
</tr>
</thead>
<tbody>
<tr>
<td>[64x44]Index of grammar productions 1745</td>
</tr>
<tr>
<td>[64x630]string-literal</td>
</tr>
<tr>
<td>[64x153]type-constraint</td>
</tr>
<tr>
<td>[64x177]translation-unit</td>
</tr>
<tr>
<td>[64x201]trailing-return-type</td>
</tr>
<tr>
<td>[64x237]throw-expression</td>
</tr>
<tr>
<td>[64x297]template-parameter-list</td>
</tr>
<tr>
<td>[64x512]template-argument-list</td>
</tr>
<tr>
<td>[64x572]template-argument</td>
</tr>
<tr>
<td>[64x799]©</td>
</tr>
<tr>
<td>[64x249]text-line</td>
</tr>
<tr>
<td>[64x261]template-parameters</td>
</tr>
<tr>
<td>[64x464]template-head</td>
</tr>
<tr>
<td>[94x117], 3, 73, 74, 91, 111, 116, 123–125, 171, 174, 176, 176, 177, 183, 230, 351, 354, 355, 386, 395, 396, 1622</td>
</tr>
<tr>
<td>type-name, 34, 45, 49, 50, 97, 121, 155, 158, 167, 168, 169, 177, 241, 265, 365, 388, 389, 1621, 1646</td>
</tr>
<tr>
<td>type-parameter, 169, 183, 185, 347, 348, 350, 370, 386, 392, 430, 1628</td>
</tr>
<tr>
<td>type-parameter-key, 348, 348, 1628</td>
</tr>
<tr>
<td>type-requirement, 109, 109, 1616</td>
</tr>
<tr>
<td>type-specifier, 53, 146, 147, 152, 158, 165, 166, 167, 170, 171, 174, 182, 250, 252, 1621, 1654</td>
</tr>
<tr>
<td>type-specifier-seq, 124, 165, 166, 170, 171, 174, 211, 1621</td>
</tr>
<tr>
<td>typename-specifier, 113, 166, 352, 385, 385, 1628</td>
</tr>
<tr>
<td>ud-suffix, 24, 24, 25, 344, 1614</td>
</tr>
<tr>
<td>unary-expression, 120, 265, 339, 396, 1617</td>
</tr>
<tr>
<td>unary-operator, 120, 339, 1617</td>
</tr>
<tr>
<td>universal-character-name, 10, 11, 11, 12, 15, 20, 23, 1610</td>
</tr>
<tr>
<td>unnamed-name-space-definition, 214, 216, 1624</td>
</tr>
<tr>
<td>unqualified-id, 42, 48, 50, 97, 97, 98, 121, 122, 151, 177, 208, 210, 216, 221, 283, 348, 351, 372, 391, 407, 1615, 1637</td>
</tr>
<tr>
<td>unsigned-suffix, 17, 1612</td>
</tr>
<tr>
<td>user-defined-character-literal, 23, 24, 1614</td>
</tr>
<tr>
<td>user-defined-floating-point-literal, 23, 24, 1614</td>
</tr>
<tr>
<td>user-defined-integer-literal, 23, 24, 1614</td>
</tr>
<tr>
<td>user-defined-literal, 11, 23, 24, 95, 1614</td>
</tr>
<tr>
<td>user-defined-string-literal, 23, 24, 25, 344, 1614</td>
</tr>
<tr>
<td>using-declarator, 34, 45, 220, 221, 224–226, 238, 370, 1624</td>
</tr>
<tr>
<td>using-declarator-list, 220, 1624</td>
</tr>
<tr>
<td>using-directive, 27, 33, 35, 36, 38, 43, 45, 46, 48, 50, 52, 98, 156, 215, 218, 218, 219, 220, 311, 1624</td>
</tr>
<tr>
<td>using-enum-declaration, 27, 213, 213, 214, 1624</td>
</tr>
<tr>
<td>va-opt-replacement, 450, 450, 451, 1630</td>
</tr>
<tr>
<td>virt-specifier, 251, 253, 279, 1626</td>
</tr>
<tr>
<td>virt-specifier-seq, 204, 251, 253, 1626</td>
</tr>
<tr>
<td>yield-expression, 122, 138, 138, 142, 207, 1618</td>
</tr>
</tbody>
</table>
Index of library headers

The bold page number for each entry refers to the page where the synopsis of the header is shown.

<algorithm>, 492, 494, 1029, 1647
<any>, 492, 604, 1642
<array>, 492–494, 579, 818, 818, 821, 957, 1647, 1672, 1673
<assert.h>, 471, 550, 1659, 1663
<atomic>, 470, 492, 1520, 1647
<atomics>, 1682
<barrier>, 492, 1593, 1638
<bt>, 492, 493, 1150, 1638
<bitset>, 609
<cassert>, 471, 550, 1659
<ctype>, 780, 1324
<cerrno>, 482, 550, 554
<cfenv>, 1141, 1142, 1647
<cmath>, 492, 502, 502
<charconv>, 494, 718, 1638, 1642
<chrono>, 492, 1225, 1647
<cinttypes>, 1483, 1485, 1647
<cinttypes>, 54, 492, 501, 501, 1667
<concepts>, 467, 1353, 1659
<cmath>, 493, 1209, 1217, 1638, 1663
<codecvt>, 1647, 1676
<compare>, 133, 491, 520, 1638
<complex>, 492, 1142, 1143, 1638, 1663
<complex.h>, 1659, 1663, 1663, 1664
<concepts>, 492, 535, 1638
<condition_variable>, 1583, 1647
<coroutine>, 493, 527, 528, 1638
<csetjmp>, 482, 532, 532, 1660
<csignal>, 532, 533
<cstdlib>, 183, 482, 532, 532
<cstdlib>, 123, 132, 489, 492, 1659, 1660
<cstdlib>, 71, 502, 1485, 1530, 1537, 1647
<cstdlib>, 504, 1358, 1359, 1365, 1424, 1482, 1483, 1483, 1659
<cstdlib>, 85, 470, 489, 490, 503, 504, 507, 532, 614, 630, 783, 1140, 1190, 1217, 1334, 1659, 1660, 1664
<cstring>, 255, 467, 781, 1659, 1667, 1670
<cctime>, 1313, 1313, 1316, 1659
<ctype.h>, 780, 1663
<cuchar>, 482, 783, 783, 1647, 1659
<cwchar>, 482, 742, 781, 782, 783, 1659
<cwchar>, 482, 780
deque>, 492, 493, 818, 819, 957
<errno.h>, 550, 1663
<exception>, 494, 515
<execution>, 493, 717, 718, 1642
<errno.h>, 1142, 1663
<filesystem>, 492, 493, 1439, 1642
<float.h>, 502, 1663
<format>, 493, 721, 1638
<forward_list>, 492, 493, 818, 820, 957, 1647
<fstream>, 1421, 1421, 1481
<functional>, 492–494, 666
<future>, 1596, 1647
<initializer_list>, 519, 1647
<inttypes.h>, 1485, 1663
<iomanip>, 494, 1383, 1402, 1404
<iostream>, 1359
<iostream>, 1359, 1357
<iostream>, 388, 1358, 1358, 1365
<isoc99.h>, 469, 1659, 1663, 1663, 1664
<iostream>, 492, 1383, 1384
<iterator>, 492–494, 901, 957, 965, 1673
<limits>, 493, 1592, 1638
<limits>, 492, 494
<limits.h>, 501, 1663
<list>, 492, 493, 818, 820, 957
<locale>, 492, 1315, 1316, 1677
<locale.h>, 1353, 1663
<map>, 492, 493, 846, 847, 957
<math.h>, 1217, 1663
<memory>, 492–494, 614, 615, 1134, 1674
<memory_resource>, 493, 653, 1642
<mutex>, 494, 1565, 1647
<new>, 62, 493, 504
<numbers>, 493, 1224, 1638
<numeric>, 493, 494, 1121
<optional>, 493, 581, 1642
<ostream>, 492, 1383, 1394
<queue>, 886, 887
<random>, 1154, 1647
<ranges>, 494, 579, 960, 965, 1638, 1672, 1673
<ratio>, 713, 1647
<regex>, 493, 957, 1488, 1488, 1647
<scoped_allocator>, 492, 662, 1647
<semaphore>, 494, 1591, 1638
<set>, 492, 493, 846, 848, 957
<setjmp.h>, 532, 1663
<shared_mutex>, 494, 1566, 1645
<signal.h>, 533, 1663
<source_location>, 494, 513, 1638
<span>, 494, 894, 894, 957, 1638
<sstream>, 1407, 1407
<stack>, 886, 887
<stdalign.h>, 1659, 1663, 1663, 1664
<stdarg.h>, 532, 1663
<stdbool.h>, 1653, 1659, 1663, 1663, 1664
<stdio.h>, 1483, 1663
<stdlib.h>, 490, 1663, 1664
<stop_token>, 493, 1554, 1638
<streambuf>, 1375, 1375
<string>, 492–494, 741, 744, 957
<string.h>, 781, 1663
<string_view>, 492–494, 770, 957, 1642
<strstream>, 1664
<syncstream>, 494, 1433, 1433, 1638
<system_error>, 552, 554, 1647
<tgmath.h>, 1659, 1663, 1663, 1664
<thread>, 493, 1558, 1647
<time.h>, 1313, 1663
<tuple>, 492–494, 563, 571, 579, 1647, 1672, 1673
<type_traits>, 492–494, 689, 1647, 1672
<typeindex>, 715, 1647
<typeinfo>, 116, 511
<wchar.h>, 783, 1663
<unordered_map>, 492–494, 864, 864, 957, 1647
<unordered_set>, 492, 493, 864, 865, 957, 1647
<utility>, 473, 492–494, 561, 579, 1647, 1664, 1672, 1673
<vector>, 492, 492, 492, 1190, 1192
<variant>, 494, 593, 1642, 1673
<vector>, 492, 493, 818, 821, 957
<version>, 492, 492, 1638
<wchar.h>, 782, 1663
<wctype.h>, 780, 1663
Index of library names

Symbols

_EXIT, 489, 503
_IOFBF, 1482
_IOLBF, 1482
_IONBF, 1482
_alignas_is_defined, 1663
_BOOL_TRUE_FALSE_are_defined, 1663
__cpp_lib_addressof_constexpr, 492
__cpp_lib_allocator_traits_is_always_equal, 1663
__cpp_lib_any, 492
__cpp_lib_arrary_constexpr, 492
__cpp_lib_as_const, 492
__cpp_lib_assume_aligned, 492
__cpp_lib_atomic_flag_test, 492
__cpp_lib_atomic_float, 492
__cpp_lib_atomic_is_always_lock_free, 492
__cpp_lib_atomic_lock_free_type_aliases, 492
__cpp_lib_atomic_ref, 492
__cpp_lib_atomic_shared_ptr, 492
__cpp_lib_atomic_value_initialization, 492
__cpp_lib_atomic_lock_free_type_aliases, 492
__cpp_lib_atomic_wait, 492
__cpp_lib_addressof, 492
__cpp_lib_barrier, 492
__cpp_lib_bind_front, 492
__cpp_lib_bit_cast, 492
__cpp_lib_bitops, 492
__cpp_lib_bool_constant, 492
__cpp_lib_bound_array_traits, 492
__cpp_lib_boyer_moore_searcher, 492
__cpp_lib_byte, 492
__cpp_lib_char8_t, 492
__cpp_lib_chrono, 492
__cpp_lib_chrono_udls, 492
__cpp_lib_clamp, 492
__cpp_lib_complex_udls, 492
__cpp_lib_concepts, 492
__cpp_lib_constexpr_algorithms, 492
__cpp_lib_constexpr_complex, 492
__cpp_lib_constexpr_dynamic_alloc, 492
__cpp_lib_constexpr_functional, 492
__cpp_lib_constexpr_iterator, 492
__cpp_lib_constexpr_memory, 492
__cpp_lib_constexpr_numeric, 492
__cpp_lib_constexpr_string, 492
__cpp_lib_constexpr_string_view, 492
__cpp_lib_constexpr_tuple, 492
__cpp_lib_constexpr_utility, 492
__cpp_lib_constexpr_vector, 492
__cpp_lib_coroutine, 493
__cpp_lib_destroying_delete, 493
__cpp_lib_enable_shared_from_this, 493
__cpp_lib_endian, 493
__cpp_lib_erase_if, 493
__cpp_lib_exchange_function, 493
__cpp_lib_execution, 493
__cpp_lib_filesystem, 493
__cpp_lib_format, 493
__cpp_lib_gcd_lcm, 493
__cpp_lib_generic_associative_lookup, 493
__cpp_lib_generic_unordered_lookup, 493
__cpp_lib_hardware_interference_size, 493
__cpp_lib_has_unique_object_representations, 493
__cpp_lib_hypot, 493
__cpp_lib_incomplete_container_elements, 493
__cpp_lib_int_pow2, 493
__cpp_lib_integer_comparison_functions, 493
__cpp_lib_integer_sequence, 493
__cpp_lib_integral_constant_callable, 493
__cpp_lib_interpolate, 493
__cpp_lib_invoke, 493
__cpp_lib_is_aggregate, 493
__cpp_lib_is_constant_evaluated, 493
__cpp_lib_is_final, 493
__cpp_lib_is_invocable, 493
__cpp_lib_is_layout_compatible, 493
__cpp_lib_is_nothrow_convertible, 493
__cpp_lib_is_null_pointer, 493
__cpp_lib_is_pointer_interconvertible, 493
__cpp_lib_is_swappable, 493
__cpp_lib_jthread, 493
__cpp_lib_latch, 493
__cpp_lib_launder, 493
__cpp_lib_list_remove_return_type, 493
__cpp_lib_logical_traits, 493
__cpp_lib_make_from_tuple, 493
__cpp_lib_make_reverse_iterator, 493
__cpp_lib_make_unique, 493
__cpp_lib_map_try_emplace, 493
__cpp_lib_math_constants, 493
__cpp_lib_math_special_functions, 493
__cpp_lib_memory_resource, 493
__cpp_lib_node_extract, 493
__cpp_lib_nonmember_container_access, 493
__cpp_lib_not_fn, 493
__cpp_lib_null_iterators, 493
__cpp_lib_optional, 493
__cpp_lib_parallel_algorithm, 494
__cpp_lib_polymorphic_allocator, 494
__cpp_lib_quoted_string_io, 494
Index of library names

- scoped_allocator_adaptor, 665
- allocate_bytes
- polymorphic_allocator, 656
- allocate_object
- polymorphic_allocator, 656
- allocate_shared, 645–647
- allocator, 629
  - allocate, 629
  - deallocate, 629
  - difference_type, 629
  - is_always_equal, 629
  - operator=, 629
  - operator==, 630
  - propagate_on_container_move_assignment, 629
  - size_type, 629
  - allocator_arg, 629
  - value_type, 629
- allocator_traits, 627
  - allocate, 628
  - const_pointer, 628
  - const_void_pointer, 628
  - construct, 629
  - deallocate, 628
  - destroy, 629
  - difference_type, 628
  - is_always_equal, 628
  - max_size, 629
  - pointer, 628
  - propagate_on_container_copy_assignment, 628
  - propagate_on_container_move_assignment, 628
  - propagate_on_container_swap, 628
  - rebind_alloc, 628
  - select_on_container_copy_construction, 629
  - size_type, 628
  - void_pointer, 628
- allocator_type
  - basic_string, 747
- alpha
  - gamma_distribution, 1179
- always_noconv
  - codecvt, 1329
- ambiguous
  - local_info, 1298
  - ambiguous_local_time, 1296
  - constructor, 1296
- any
  - constructor, 605, 606
  - destructor, 606
  - emplace, 607
  - has_value, 607
  - operator=, 606, 607
  - reset, 607
  - swap, 607, 608
  - type, 608
  - any (member)
    - bitset, 613
  - any_cast, 608
  - any_of, 1067
- append
  - basic_string, 757
  - path, 1450
- apply, 578
  - valarray, 1199
  - arg, 1150
  - basic_format_context, 734
  - complex, 1148
- argument_type
  - zombie, 482
- array, 821, 824
  - begin, 821
  - data, 823
  - end, 821
  - fill, 823
  - get, 824
  - max_size, 821
  - size, 821, 823
  - swap, 823
- arrive
  - barrier, 1595
  - arrive_and_drop
    - barrier, 1595
  - arrive_and_wait
    - barrier, 1595
    - latch, 1593
- as_bytes, 900
- as_const, 565
- as_writable_bytes, 900
- asctime, 1313
- asin, 1209
  - complex, 1148
  - valarray, 1201
- asinf, 1209
- asinh, 1209
  - complex, 1149
- asinhf, 1209
- asinhl, 1209
- asinl, 1209
- assert, 550
- assign
  - basic_regex, 1500, 1501
  - basic_string, 757, 758
  - directory_entry, 1463
  - error_code, 556
  - error_condition, 558
  - path, 1449
- assignable_from, 539
- assoc_laguerre, 1218
- assoc_laguerref, 1218
- assoc_laguerrel, 1218
- assoc_legendre, 1219

N4861
assoc_legendref, 1219
assoc_legendrel, 1219
assume_aligned, 624
async, 1605
at
  basic_string, 756
  basic_string_view, 775
  map, 852
  unordered_map, 871
at_quick_exit, 470, 489, 504
atan, 1209
  complex, 1148
  valarray, 1201
atan2, 1209
  valarray, 1201
atan2f, 1209
atan2l, 1209
atanf, 1209
atanh, 1209
  complex, 1149
atanhf, 1209
atanhl, 1209
atexit, 85, 470, 489, 503
atof, 489
atoi, 489
atol, 489
atoll, 489
atomic, 1533, 1534
  compare_exchange_strong, 1535
  compare_exchange_weak, 1535
  constructor, 1534
  exchange, 1535
  is_always_lock_free, 1535
  is_lock_free, 1535
  load, 1535
  notify_all, 1537
  notify_one, 1537
  operator type, 1535
  operator=, 1535
  store, 1535
  value_type, 1533
  wait, 1537
atomic<floating-point>, 1539
  compare_exchange_strong, 1535
  compare_exchange_weak, 1535
  constructor, 1534
  exchange, 1535
  fetch_add, 1541
  fetch_and, 1541
  fetch_or, 1539
  fetch_sub, 1539
  fetch_xor, 1539
  is_always_lock_free, 1535
  is_lock_free, 1535
  load, 1535
  notify_all, 1537
  notify_one, 1537
  operator floating-point, 1535
  operator++, 1541
  operator=, 1541
  operator--, 1535
  store, 1535
  wait, 1537
atomic<integral>, 1537
  compare_exchange_strong, 1535
  compare_exchange_weak, 1535
  constructor, 1534
  exchange, 1535
  fetch_add, 1539
  fetch_and, 1539
  fetch_or, 1539
  fetch_sub, 1539
  fetch_xor, 1539
  is_always_lock_free, 1535
  is_lock_free, 1535
  load, 1535
  notify_all, 1537
  notify_one, 1537
  operator integral, 1535
  operator++, 1543
  operator+=, 1539
  operator-=, 1543
  operator=, 1539
  operator*=, 1539
  operator/=, 1539
  operator*, 1539
  operator+=, 1539
  operator-=, 1539
  operator=, 1535
  store, 1535
  wait, 1537
atomic<shared_ptr<T>>, 1544
  compare_exchange_strong, 1545
  compare_exchange_weak, 1545
  constructor, 1544
  exchange, 1545
  is_always_lock_free, 1535
  is_lock_free, 1535
  load, 1545
  notify_all, 1546
  notify_one, 1546
  operator shared_ptr<T>, 1545
  operator=, 1545
  store, 1544
  wait, 1545
atomic<T*>, 1541, 1542
  compare_exchange_strong, 1535
  compare_exchange_weak, 1535
  constructor, 1534
  exchange, 1535
  fetch_add, 1542
  fetch_sub, 1542
  is_always_lock_free, 1535
  is_lock_free, 1535
  load, 1535
  notify_all, 1537
  notify_one, 1537
  operator T*, 1535
  operator++, 1543
  operator+=, 1539, 1541, 1543
  operator-=, 1539, 1541, 1543
  operator=, 1543
Index of library names
operator $\text{floating-point}$, 1528
operator+=, 1532
operator=-, 1532
operator=, 1528
required_alignment, 1528
store, 1528
atomic_ref<integral>, 1530
compare_exchange_strong, 1529
compare_exchange_weak, 1529
constructor, 1528
exchange, 1528
fetch_add, 1531
fetch_and, 1531
fetch_or, 1531
fetch_sub, 1531
fetch_xor, 1531
is_always_lock_free, 1528
is_lock_free, 1528
load, 1528
operator integral, 1528
operator++, 1533
operator+=, 1531
operator=-, 1531
operator=, 1533
operator=, 1528
operator=, 1531
operator=, 1531
operator=, 1531
required_alignment, 1528
store, 1528
atomic_ref<T>, 1532
compare_exchange_strong, 1529
compare_exchange_weak, 1529
constructor, 1528
exchange, 1528
fetch_add, 1533
fetch_sub, 1533
is_always_lock_free, 1528
is_lock_free, 1528
load, 1528
operator T*, 1528
operator++, 1533
operator+=, 1533
operator=-, 1533
operator=, 1528
operator=, 1531
operator=, 1531
required_alignment, 1528
store, 1528
atomic_ref<T>
notify_all, 1529
notify_one, 1529
wait, 1529
atomic_schar, 1524
atomic_short, 1524
ATOMIC_SHORT_LOCK_FREE, 1526
atomic_signal_fence, 1550
atomic_signed_lock_free, 1524
atomic_size_t, 1524
atomic_store, 1535
shared_ptr, 1675
atomic_store_explicit, 1535
shared_ptr, 1675
atomic_thread_fence, 1550
atomic_uchar, 1524
atomic_uint, 1524
atomic_uint16_t, 1524
atomic_uint32_t, 1524
atomic_uint64_t, 1524
atomic_uint8_t, 1524
atomic_uint_fast16_t, 1524
atomic_uint_fast32_t, 1524
atomic_uint_fast64_t, 1524
atomic_uint_fast8_t, 1524
atomic_uint_least16_t, 1524
atomic_uint_least32_t, 1524
atomic_uint_least64_t, 1524
atomic_uint_least8_t, 1524
atomic_uintmax_t, 1524
atomic_uintptr_t, 1524
atomic_ullong, 1524
atomic_ulong, 1524
atomic_unsigned_lock_free, 1524
atomic_ushort, 1524
atomic_uchar, 1524
ATOMIC_VAR_INIT, 1682
atomic_wchar_t, 1524
ATOMIC_WCHAR_T_LOCK_FREE, 1526
auto_ptr
zombie, 481
auto_ptr_ref
zombie, 481
await_ready
suspend_always, 531
suspend_never, 531
await_resume
suspend_always, 531
suspend_never, 531
await_suspend
suspend_always, 531
suspend_never, 531
awk
syntax_option_type, 1492, 1493

B
b
caucchy_distribution, 1184
extreme_value_distribution, 1181
uniform_int_distribution, 1174
uniform_real_distribution, 1175
weibull_distribution, 1180
back
basic_string, 756
basic_string_view, 775
span, 899
view_interface, 973
back_insert_iterator, 936
constructor, 936
operator*, 936
basic_ostringstream<wchar_t>, 1407
basic_osyncstream, 1356, 1435
   constructor, 1436
set_emit_on_sync, 1436, 1437
basic_regex, 1488, 1497, 1517
   assign, 1500, 1501
   constructor, 1499, 1500
   flag_type, 1501
getloc, 1501
imbue, 1501
mark_count, 1501
operator=, 1500
swap, 1501
basic_streambuf, 1356, 1376
   constructor, 1377
destructor, 1378
eback, 1379
egptr, 1379
epptr, 1380
gbump, 1379
getloc, 1378
gptr, 1379
imbue, 1380
in_avail, 1378
operator=, 1379
overflow, 1382
pbackfail, 1382
pbase, 1380
pbump, 1380
pptr, 1380
pubimbue, 1378
pubseekoff, 1378
pubseekpos, 1378
pubsetbuf, 1378
pubsync, 1378
sbumpc, 1378
seekoff, 1380
seekpos, 1380
setbuf, 1380, 1413
setg, 1379
setp, 1380
sgetc, 1378
sgetn, 1378
showmanyc, 1381, 1424
sngetc, 1378
spbackcx, 1379
sputc, 1379
sputn, 1379
sungetc, 1379
swap, 1379
sync, 1380
uflow, 1381
underflow, 1381
xsgetn, 1381
xsputn, 1382
basic_streambuf<char>, 1375
basic_streambuf<wchar_t>, 1375
basic_string, 747, 765, 1407, 1676
   allocator_type, 747
append, 757
assign, 757, 758
at, 756
back, 756
begin, 754
c_str, 762
capacity, 755
cbegin, 754
cend, 755
clear, 756
compare, 764
const_iterator, 747
const_pointer, 747
const_reference, 747
const_reverse_iterator, 747
constructor, 752, 753
copy, 762
crbegin, 755
crend, 755
data, 762
difference_type, 747
empty, 756
end, 755
ends_with, 765
erase, 760, 768
erase_if, 768
find, 763
find_first_not_of, 763
find_first_of, 763
find_last_not_of, 763
find_last_of, 763
front, 756
get_allocator, 763
getline, 767, 768
insert, 758–760
iterator, 747
length, 755
max_size, 755
operator basic_string_view, 763
operator+, 765, 766
operator+=, 756
operator<<, 767
operator<=, 754
operator>>, 767
operator[], 756
pointer, 747
pop_back, 760
push_back, 757
rbegin, 755
reference, 747
rend, 755
replace, 760–762
reserve, 755, 1676
resize, 755
reverse_iterator, 747
rfind, 763
shrink_to_fit, 755
size, 755
size_type, 747

Index of library names 1756
starts_with, 764
substr, 763
swap, 762, 767
traits_type, 747
value_type, 747
basic_string_view, 771
at, 775
back, 775
begin, 774
cbegin, 774
cend, 774
compare, 776
const_iterator, 771, 774
const_pointer, 771
const_reference, 771
const_reverse_iterator, 771
constructor, 773
copy, 775
crbegin, 774
crend, 774
data, 775
difference_type, 771
empty, 774
dend, 774
ends_with, 776
find, 777
find_first_not_of, 777
find_first_of, 777
find_last_not_of, 778
find_last_of, 777
front, 775
iterator, 771
length, 774
max_size, 774
operator<<, 779
operator<=, 779
operator==, 778
operator[], 774
pointer, 771
rbegin, 774
reference, 771
remove_prefix, 775
remove_suffix, 775
rend, 774
reverse_iterator, 771
rfind, 777
size, 774
size_type, 771
starts_with, 776
substr, 775
swap, 775
traits_type, 771
value_type, 771
basic_stringbuf, 1356, 1407
constructor, 1409
get_allocator, 1411
operator=, 1410
overflow, 1412
pbackfail, 1411
seekoff, 1412
seekpos, 1413
str, 1411
swap, 1410
underflow, 1411
view, 1411
basic_stringbuf<char>, 1407
basic_stringbuf<wchar_t>, 1407
basic_stringstream, 1356, 1418
constructor, 1419, 1420
ddbuf, 1420
str, 1420, 1421
swap, 1420
view, 1420
basic_stringstream<char>, 1407
basic_stringstream<wchar_t>, 1407
basic_syncbuf, 1356, 1433
constructor, 1434
destructor, 1434
emit, 1435
get_allocator, 1435
get_wrapped, 1435
operator=, 1434
set_emit_on_sync, 1435
swap, 1434, 1435
sync, 1435
before

type_info, 512
before_begin
forward_list, 830
begin, 519
array, 821
basic_format_parse_context, 733
basic_string, 754
basic_string_view, 774
common_view, 1015
directory_iterator, 1467
drop_view, 1003
drop_while_view, 1004
elements_view, 1018
filter_view, 990
initializer_list, 519
iota_view, 981
join_view, 1005
match_results, 1506
path, 1456
recursive_directory_iterator, 1470
reverse_view, 1017
single_view, 979
span, 899
split_view, 1010
split_view::outer_iterator::value_type, 1013
subrange, 976
sys_info, 1297
take_view, 999
take_while_view, 1001
transform_view, 993

Index of library names 1757
Index of library names

bitset, 609
all, 613
any, 613
constructor, 611
count, 613
flip, 612
none, 613
operator<<, 613, 614
operator<<=, 612
operator==, 613
operator>>, 613, 614
operator>>=, 612
operator[], 613, 614
operator&=, 614
operator&&, 611
operator~, 614
operator~, 612
operator|, 614
operator|=, 611
reset, 612
set, 612
size, 613
test, 613
to_string, 613
to_ullong, 613
to_ulong, 613
bool_constant, 696
boolalpha, 1372
borrowed_range, 969
boyer_moore_horspool_searcher, 687
constructor, 687
operator(), 688
boyer_moore_searcher, 686
constructor, 687
operator(), 687
bsearch, 489, 1140
btowc, 781
bucket
unordered associative containers, 816
bucket_count
unordered associative containers, 816
bucket_size
unordered associative containers, 816
BUFSIZ, 1482
byte, 488
operator<<, 491
operator<<=, 491
operator>>, 491
operator>>=, 491
operator&=, 491
operator~, 492
operator~, 492
operator|, 491
operator|=, 491
to_integer, 492
byte_string
Index of library names

unordered associative containers, 817

cend(const Ck), 958

cerr, 1359

CHAR_BIT, 501

char_class_type

regex_traits, 1496

CHAR_MAX, 501

CHAR_MIN, 501

char_traits, 741–743

char_type, 741

int_type, 741

state_type, 741

char_type

basic_format_context, 733

basic_format_parse_context, 732

char_traits, 741

chars_format, 718

fixed, 718

general, 718

hex, 718

scientific, 718

check_arg_id

basic_format_parse_context, 733

chi_squared_distribution, 1183

constructor, 1183

n, 1183

result_type, 1183

choose, 1225

earliest, 1225

latest, 1225

chrono, 1225

cin, 1359

clamp, 1118

classic

locale, 1322

classic_table
c_type<char>, 1327

clear

atomic_flag, 1549

basic_ios, 1372

basic_string, 756

error_code, 556

error_condition, 558

forward_list, 832

ordered associative containers, 805

path, 1451

unordered associative containers, 815

clearerr, 1482

clock, 1313

clock_cast, 1261

clock_t, 1313

clock_time_conversion, 1259

operator(), 1259–1261

CLOCKS_PER_SEC, 1313

clog, 1359

close

basic_filebuf, 1424

basic_fstream, 1432

basic_ifstream, 1428

wstring_convert, 1679

C

c16rtomb, 782

c32rtomb, 782

c8rtomb, 782, 784

c_encoding

weekday, 1269

c_str

basic_string, 762

path, 1452

cacos

c_complex

cacosh

c_complex

catan

c_complex

c_tanh

category

error_code, 557

error_condition, 558

locale, 1318

cauchy_distribution, 1183

a, 1184

b, 1184

constructor, 1184

result_type, 1183

cbefore_begin

forward_list, 830

cbegin

basic_string, 754

basic_string_view, 774

tzdb_list, 1294

unordered associative containers, 817

cbegin(const Ck), 958

cbrt, 1209

cbrtf, 1209

cbrtl, 1209

cceil, 1209

duration, 1245

time_point, 1250

cceilf, 1209

cceil, 1209

cend

basic_string, 755

basic_string_view, 774

tzdb_list, 1295
basic_ofstream, 1430
messages, 1352
cmp_equal, 565
cmp_greater, 566
cmp_greater_equal, 566
cmp_less, 566
cmp_less_equal, 566
cmp_not_equal, 566
code
   future_error, 1597
   system_error, 500
codecvt, 1328
   always_noconv, 1329
   do_always_noconv, 1331
   do_encoding, 1331
   do_in, 1329
   do_length, 1331
   do_max_length, 1331
   do_out, 1329
   do_unshift, 1330
   encoding, 1329
   in, 1329
   length, 1329
   max_length, 1329
   out, 1329
   unshift, 1329
codecvt_byname, 1331
codecvt_mode, 1676
codecvt_utf16, 1676, 1677
codecvt_utf8, 1676, 1677
codecvt_utf8_utf16, 1676, 1677
collate, 1340
   compare, 1341
   do_compare, 1341
   do_hash, 1341
   do_transform, 1341
   hash, 1341
   syntax_option_type, 1492, 1493, 1518
   transform, 1341
collate_byname, 1341
combine
locale, 1321
common_comparison_category, 524
common_comparison_category_t, 520
common_iterator, 943
   constructor, 945
   iter_move, 947
   iter_swap, 947
   operator*, 945
   operator++, 946
   operator-, 946
   operator->, 945
   operator=, 945
   operator==, 946
common_range, 972
common_reference, 709
common_reference_t, 692
common_reference_with, 538
common_type, 709, 1240, 1243, 1244
common_type_t, 692
common_view, 1015
   base, 1015
   begin, 1015
   constructor, 1016
   end, 1015
   size, 1015
   common_with, 538
comp
   map::value_compare, 849
   multimap::value_compare, 854
comp_ellint_1, 1219
comp_ellint_1f, 1219
comp_ellint_1l, 1219
comp_ellint_2, 1219
comp_ellint_2f, 1219
comp_ellint_2l, 1219
comp_ellint_3, 1220
comp_ellint_3f, 1220
comp_ellint_3l, 1220
compare
   basic_string, 764
   basic_string_view, 776
   collate, 1341
   path, 1453
   sub_match, 1502
compare_exchange_strong
   atomic, 1535
   atomic\langle floating-point\rangle, 1535
   atomic\langle integral\rangle, 1535
   atomic\langle shared_ptr\langle T\rangle\rangle, 1545
   atomic\langle T\rangle*, 1535
   atomic\langle weak_ptr\langle T\rangle\rangle, 1547
   atomic_ref, 1529
   atomic_ref\langle floating-point\rangle, 1529
   atomic_ref\langle integral\rangle, 1529
   atomic_ref\langle T\rangle*, 1529
compare_exchange_weak
   atomic, 1535
   atomic\langle floating-point\rangle, 1535
   atomic\langle integral\rangle, 1535
   atomic\langle shared_ptr\langle T\rangle\rangle, 1545
   atomic\langle T\rangle*, 1535
   atomic\langle weak_ptr\langle T\rangle\rangle, 1547
   atomic_ref, 1529
   atomic_ref\langle floating-point\rangle, 1529
   atomic_ref\langle integral\rangle, 1529
   atomic_ref\langle T\rangle*, 1529
compare_partial_orderFallback, 527
compare_strong_orderFallback, 527
compare_three_way, 675
compare_weak_orderFallback, 527
complex, 1144
   abs, 1148
   acos, 1148
   acosh, 1149
   arg, 1148
   asin, 1148
   asinh, 1149

Index of library names 1760
atan, 1148
atanh, 1149
cacos, 1148
cacosh, 1149
casin, 1148
casinh, 1149
cat, 1148
catanh, 1149
conj, 1148
constructor, 1146
cos, 1149
cosh, 1149
exp, 1149
imag, 1146, 1148
log, 1149
log10, 1149
norm, 1148
operator"i", 1150
operator"if", 1150
operator"il", 1150
operator*, 1147
operator*, 1146, 1147
operator+, 1147
operator+, 1146
operator-, 1147
operator-=, 1146, 1147
operator/, 1147
operator<<, 1148
operator<<, 1146, 1147
operator>>, 1147
polar, 1148
pow, 1149
proj, 1148
real, 1146, 1148
sin, 1149
sinh, 1149
sqrt, 1149
tan, 1149
tanh, 1149
value_type, 1144
concat
cat, 1450, 1451
condition_variable, 1584
condition_variable_any, 1587
conditional_t, 692
conj, 1150
complex, 1148
conjunction, 711
conjunction_v, 696
const_iterator
basic_format_parse_context, 732
basic_string, 747
basic_string_view, 771, 774
const_local_iterator
unordered associative containers, 809
const_mem_fun1_ref_t
zombie, 481
const_mem_fun1_t
zombie, 481
const_mem_fun_ref_t
zombie, 481
const_mem_fun_t
zombie, 481
const_pointer
allocator_traits, 628
basic_string, 747
basic_string_view, 771
scoped_allocator_adaptor, 662
const_pointer_cast
shared_ptr, 649
const_reference
basic_string, 747
basic_string_view, 771
const_reverse_iterator
basic_string, 747
basic_string_view, 771
const_void_pointer
allocator_traits, 628
scoped_allocator_adaptor, 662
construct
allocator_traits, 629
polymorphic_allocator, 657
scoped_allocator_adaptor, 665
construct_at, 1139
constructible_from, 541
consume
memory_order, 1524
contains
ordered associative containers, 805
unordered associative containers, 816
contiguous_iterator, 920
contiguous_iterator_tag, 928
contiguous_range, 971
converted
wstring_convert, 1679
convertible_to, 538
copy, 1078, 1079
basic_string, 762
basic_string_view, 775
path, 1471
copy_backward, 1080
copy_constructible, 542
copy_file, 1473
Index of library names

copy_if, 1079
copy_n, 1079
copy_options, 1459
copy_symmlink, 1473
copyable, 545
copyfmt
  basic_ios, 1371
copysign, 1209
copysignf, 1209
copysignl, 1209
coroutine_handle, 528
  address, 529
  constructor, 529
  destroy, 530
done, 530
from_address, 529
from_promise, 529
hash, 530
operator bool, 529
operator+=, 530
operator(), 530
operator==, 530
operator++, 530
operator=, 529
operator-, 530
promise, 530
resume, 530
coroutine_handle<noop_coroutine_promise>, 530
  address, 531
  destroy, 531
done, 531
operator bool, 531
operator+=, 531
operator(), 531
operator==, 531
promise, 531
resume, 531

cos, 1209
  complex, 1149
  valarray, 1201
cosf, 1209
cosh, 1209
  complex, 1149
  valarray, 1201
coshf, 1209
coshl, 1209
cosl, 1209
count, 1072
  bitset, 613
  counted_iterator, 949
duration, 1242
  ordered associative containers, 805
  unordered associative containers, 816
count_down
  latch, 1593
count_if, 1072
counted_iterator, 947
  base, 949
  constructor, 949
count, 949
  iter_move, 951

iter_swap, 951
operator*, 949
operator+, 950
operator++, 949, 950
operator+=, 950
operator-, 950, 951
operator-=, 951
operator<=>, 951
operator*, 949
operator==, 951
operator=, 950
counting_semaphore
  acquire, 1592
  constructor, 1591
  max, 1591
  release, 1591
try_acquire, 1592
try_acquire_for, 1592
try_acquire_until, 1592
countl_one, 1152
countl_zero, 1152
countr_one, 1153
countr_zero, 1152
cout, 1359
crbegin
  basic_string, 755
  basic_string_view, 774
crbegin(const C& c), 958
create_directories, 1473
create_directory, 1474
create_directory_symmlink, 1474
create_hard_link, 1474
create_symlink, 1474
cref
  reference_wrapper, 671
crend
  basic_string, 755
  basic_string_view, 774
crend(const C& c), 958
cshift
  valarray, 1199
cftime, 1313
cctype, 1323
do_is, 1324
do_narrow, 1325
do_scan_not, 1324
do_tolower, 1325
do_toupper, 1325
do_widen, 1325
is, 1324
  narrow, 1324
  scan_is, 1324
  scan_not, 1324
tolower, 1324
toupper, 1324
  widen, 1324
cctype<char>, 1326
  classic_table, 1327
constructor, 1326
ctype<char>, 1326
destructor, 1326
do_narrow, 1327
do_tolower, 1327
do_toupper, 1327
do_widen, 1327
is, 1327
narrow, 1327
scan_is, 1327
scan_not, 1327
table, 1327
tolower, 1327
toupper, 1327
widen, 1327
cctype_base, 1323
do_scan_is, 1324
cctype_byname, 1325
curr_symbol
  moneypunct, 1351
current_exception, 517
current_path, 1475
current_zone, 1295
tzdb, 1294
cv_status, 1583
cyl_bessel_i, 1220
cyl_bessel_if, 1220
cyl_bessel_il, 1220
cyl_bessel_j, 1220
cyl_bessel_jf, 1220
cyl_bessel_jl, 1220
cyl_bessel_k, 1220
cyl_bessel_kf, 1220
cyl_bessel_kl, 1220
cyl_neumann, 1221
cyl_neumannf, 1221
cyl_neumannl, 1221

d
  dangling, 977
data
    array, 823
    basic_string, 762
    basic_string_view, 775
    single_view, 979
    span, 899
    vector, 843
data(C& c), 959
data(initializer_list<E>), 959
data(T (&array)[N]), 959
date
  leap_second, 1304
date_order
  time_get, 1343
day, 1262
  constructor, 1262
  from_stream, 1263
  month_day, 1272
  ok, 1263
  operator unsigned, 1263
  operator"d, 1263
  operator*, 1263
  operator++, 1262
  operator*, 1263
  operator-, 1263
  operator--, 1262
  operator-=, 1263
  operator<=, 1263
  operator<=, 1263
  operator==, 1263
  year_month_day, 1279
  year_month_day_last, 1282
days, 1225
  DBL_DECIMAL_DIG, 502
  DBL_DIG, 502
  DBL_EPSILON, 502
  DBL_HAS_SUBNORM, 502
  DBL_MANT_DIG, 502
  DBL_MAX, 502
  DBL_MAX_10_EXP, 502
  DBL_MAX_EXP, 502
  DBL_MIN, 502
  DBL_MIN_10_EXP, 502
  DBL_MIN_EXP, 502
  DBL_TRUE_MIN, 502
deallocate
  allocator, 629
  allocator_traits, 628
  memory_resource, 654
  polymorphic_allocator, 656
  scoped_allocator_adaptor, 665
deallocate_bytes
  polymorphic_allocator, 656
deallocate_object
  polymorphic_allocator, 656
dec, 1374, 1400
decay, 708
  decay-copy, 465
  decay_t, 692
  DECIMAL_DIG, 502
decimal_point
  moneypunct, 1351
  numpunct, 1339
declare_no_pointers, 623
declare_reachable, 623
decval, 565
default_delete
  constructor, 631
  operator(), 631
default_error_condition
  error_category, 554, 555
  error_code, 557
default_initializable, 542
default_random_engine, 1170
default_searcher, 686
  constructor, 686
  operator(), 686

Index of library names
default_sentinel, 901
default_sentinel_t, 947
default_zone
  zoned_traits<const time_zone*>, 1300
defaultfloat, 1374
defer_lock, 1573
defer_lock_t, 1573
delete
  operator, 483, 506–509, 630
denorm_absent, 495
denorm_indeterminate, 495
denorm_min
  numeric_limits, 499
denorm_present, 495
densities
  piecewise_constant_distribution, 1188
  piecewise_linear_distribution, 1190
depth
  recursive_directory_iterator, 1469
deque, 824
  constructor, 826
  emplace, 827
  erase, 827
  erase_if, 828
  insert, 827
  push_back, 827
  push_front, 827
  resize, 826
  shrink_to_fit, 826
derived_from, 537
destroy, 1139
  allocator_traits, 629
  coroutine_handle, 530
  coroutine_handle<noop_coroutine_-promise>, 531
  polymorphic_allocator, 657
  scoped_allocator_adaptor, 665
destroy_at, 1139
destroy_n, 1140
destroying_delete, 505
destroying_delete_t, 505
destructible, 541
detach
  jthread, 1564
  thread, 1561
difference_type
  allocator, 629
  allocator_traits, 628
  basic_string, 747
  basic_string_view, 771
  pointer_traits, 622
  scoped_allocator_adaptor, 662
difftime, 1313
digits
  numeric_limits, 497
digits10
  numeric_limits, 497
directory_entry, 1462
assign, 1463
constructor, 1463
exists, 1464
file_size, 1465
hard_link_count, 1465
is_block_file, 1464
is_character_file, 1464
is_directory, 1464
is_fifo, 1464
is_other, 1464
is_regular_file, 1464
is_socket, 1464
is_symlink, 1465
last_write_time, 1465
operator const filesystem::path&, 1464
operator<=>, 1465
operator==, 1465
path, 1464
refresh, 1464
replace_filename, 1463
status, 1465
symlink_status, 1465
directory_iterator, 1465
begin, 1467
constructor, 1466, 1467
end, 1467
increment, 1467
operator++, 1467
operator=, 1467
directory_options, 1460
disable_recursion_pending
  recursive_directory_iterator, 1470
disable_sized_range, 970
disable_sized_sentinel_for, 918
discard_block_engine, 1166, 1167
constructor, 1167
result_type, 1167
discrete_distribution, 1185
constructor, 1186
probabilities, 1186
result_type, 1185
disjunction, 712
disjunction_v, 696
distance, 929–931
div, 489, 1483
div_t, 489
divides, 672
  operator(), 672
  divides<>, 672
  operator(), 672
do_allocate
  memory_resource, 654
  monotonic_buffer_resource, 662
  synchronized_pool_resource, 660
  unsynchronized_pool_resource, 660
do_always_noconv
  codecvt, 1331
do_close
  message, 1353
Index of library names

E
E2BIG, 550
EACCES, 550
EADDRINUSE, 550
EADDRNOTAVAIL, 550
EAFNOSUPPORT, 550
EAGAIN, 550
EALREADY, 550
earliest
choose, 1225
eback
basic_streambuf, 1379
EBADF, 550
EBADMSG, 550
EBUSY, 550
ec

from_chars_result, 719
to_chars_result, 719
ECANCELED, 550
ECHILD, 550
ECMAScript
syntax_option_type, 1492, 1493
ECONNABORTED, 550
ECONNREFUSED, 550
ECONNRESET, 550
EDEADLK, 550
EDESTADDRREQ, 550
EDOM, 550
EXIST, 550
EFAULT, 550
EFBIG, 550
egptr
basic_streambuf, 1379
egrep
syntax_option_type, 1492, 1493
EHOSTUNREACH, 550
EINVAL, 550
EIO
EISCONN, 550
EISDIR, 550
ellint_1, 1221
ellint_1f, 1221
ellint_1l, 1221

Index of library names 1766
ellint_2, 1221
ellint_2f, 1221
ellint_2l, 1221
ellint_3, 1221
ellint_3f, 1221
ellint_3l, 1221
ELOOP, 550
EMFILE, 550
emit
  basic_syncbuf, 1435
emit_on_flush, 1402
EMLINK, 550
emplace
  any, 607
dequ, 827
optional, 588
  ordered associative containers, 801
priority_queue, 892
  unordered associative containers, 812
variant, 599, 600
emplace_after
  forward_list, 831
emplace_front
  forward_list, 831
emplace_hint
  ordered associative containers, 801
  unordered associative containers, 812
empty
  basic_string, 756
  basic_string_view, 774
match_results, 1506
path, 1454
span, 899
subrange, 976
empty(C& c), 958
empty(initializer_list<E>), 519
empty(T (&)[N]), 958
end(C&), 957
dend(initializer_list<E>), 1297
dend(T (&)[N]), 958
dendian, 1153
  big, 1153
  little, 1153
  native, 1153
dendl, 1400, 1402
dends, 1402
dends_with
  basic_string, 756
  basic_string_view, 776
ENETDOWN, 550
ENETRESET, 550
ENETUNREACH, 550
ENFILE, 550
ENOBUFS, 550
ENODATA, 550
ENODEV, 550
ENOENT, 550
ENOEXEC, 550
ENOLCK, 550
ENOLINK, 550
ENOMEM, 550
ENOTCONN, 550
ENOTDIR, 550
ENOTEMPTY, 550
ENORECOVERY, 550
ENOTRECOVERABLE, 550
Index of library names 1767

ENOTSOCK, 550
ENOTSUP, 550
ENOTTY, 550
entropy
    random_device, 1171
ENXIO, 550
EOF, 1482
eof
    basic_ios, 1372
EOPNOTSUPP, 550
EOVERFLOW, 550
EOWNERDEAD, 550
EPERM, 550
EPIPE, 550
eptr
    basic_streambuf, 1380
EPROTO, 550
EPROTONOSUPPORT, 550
EPROTOTYPE, 550
epsilon
    numeric_limits, 497
equal, 1074
    istreambuf_iterator, 956
    strong_ordering, 523
equal_range, 1101
    ordered associative containers, 806
    unordered associative containers, 816
equal_to, 673, 675
    operator(), 673
equal_to<>, 673
    operator(), 673
equality_comparable, 544
equality_comparable_with, 544
equivalence_relation, 546
equivalent, 1475
    error_category, 554, 555
    partial_ordering, 521
    strong_ordering, 523
    weak_ordering, 522
ERANGE, 550
erase
    basic_string, 760, 768
    deque, 827
    forward_list, 834
    list, 837, 839
    ordered associative containers, 805
    unordered associative containers, 815
    vector, 844
erase_after
    forward_list, 832
tzdb_list, 1294
erase_if
    basic_string, 768
    deque, 828
    forward_list, 834
    list, 840
    map, 854
    multimap, 857
    multiset, 864
set, 861
unordered_map, 872
unordered_multimap, 877
unordered_multiset, 886
unordered_set, 882
vector, 844
erf, 1209
erfc, 1209
erfcf, 1209
erfcl, 1209
erff, 1209
erfl, 1209
EROFS, 550
errc, 552
    make_error_code, 557
    make_error_condition, 558
errno, 550
error_category, 552, 554
    constructor, 554
    default_error_condition, 554, 555
destructor, 554
    equivalent, 554, 555
    message, 555
    name, 554, 555
    operator<>, 555
    operator==, 555
error_code, 552, 555
    assign, 556
    category, 557
    clear, 556
    constructor, 556
    default_error_condition, 557
    hash, 559
    message, 557
    operator bool, 557
    operator<, 557
    operator<>, 559
    operator==, 556
    operator==, 558
    value, 557
error_condition, 552, 557
    assign, 558
    category, 558
    clear, 558
    constructor, 558
    message, 558
    operator bool, 558
    operator<, 559
    operator<>, 559
    operator==, 558
    operator==, 559
    value, 558
error_type, 1494, 1495
    regex_constants, 1494, 1495
ESPIPE, 550
ESRCH, 550
ETIME, 550
ETIMEDOUT, 550
ETXBSY, 550
EWOULD胼OCK, 550
exception, 515
  constructor, 515, 516
  destructor, 516
  operator=, 516
  what, 516
exception_ptr, 517
exceptions
  basic_ios, 1372
exchange, 564
  atomic, 1535
  atomic<floating-point>, 1535
  atomic<integral>, 1535
  atomic<shared_ptr<T>>, 1545
  atomic<T*>, 1535
  atomic<T*>, 1528
  atomic_ref<floating-point>, 1528
  atomic_ref<integral>, 1528
  atomic_ref<T*>, 1528
exclusive_scan, 1128
EXDEV, 550
execution
  par, 718
  par_unseq, 718
  seq, 718
execution::parallel_policy, 718
execution::parallel_unsequenced_policy, 718
execution::sequenced_policy, 718
execution::unsequenced_policy, 718
exists, 1475
directory_entry, 1464
exit, 82, 84, 152, 470, 489, 504, 510
EXIT_FAILURE, 489
EXIT_SUCCESS, 489
exp, 1209
complex, 1149
valarray, 1201
exp2, 1209
exp2f, 1209
exp2l, 1209
expf, 1209
expint, 1222
expintf, 1222
expintl, 1222
expired
  weak_ptr, 651
expl, 1209
expm1, 1209
expmf, 1209
expml, 1209
exponential_distribution, 1178
  constructor, 1179
  lambda, 1179
  result_type, 1178
extended
  syntax_option_type, 1492, 1493
extension
  path, 1454
extent, 703
extent_y, 695
extract
  ordered associative containers, 804
  unordered associative containers, 814
extreme_value_distribution, 1180
  a, 1181
  b, 1181
  constructor, 1181
  result_type, 1180
F
fabs, 1209
fabsf, 1209
fabsl, 1209
facet
  locale, 1319
fail
  basic_ios, 1372
failed
  ostreambuf_iterator, 957
failure
  ios_base, 1363
false_type, 696
false_name
  numpunct, 1339
fclose, 1424, 1482
fdim, 1209
fdimf, 1209
fdiml, 1209
FE_ALL_EXCEPT, 1141
FE_DFL_ENV, 1141
FE_DIVBYZERO, 1141
FE_DOWNWARD, 1141
FE_INEXACT, 1141
FE_INVALID, 1141
FE_OVERFLOW, 1141
FE_TONEAREST, 1141
FE_TOWARDZERO, 1141
FE_UPWARD, 1141
feclearexcept, 1141
fesetenv, 1141
fesetexceptflag, 1141
fesetround, 1141
feholdexcept, 1141
fenv_t, 1141
fenv_t, 1141
ferror, 1482
feraiseexcept, 1141
fesetenv, 1141
fesetexceptflag, 1141
fesetround, 1141
fetch_add
  atomic<floating-point>, 1541
  atomic<integral>, 1539
  atomic<T*>, 1542
  atomic_ref<floating-point>, 1532
atomic_ref<integral>, 1531
atomic_ref<T*>, 1533
fetch_and
atomic<integral>, 1539
atomic_ref<integral>, 1531
fetch_or
atomic<integral>, 1539
atomic_ref<integral>, 1531
fetch_sub
atomic<floating-point>, 1541
atomic<integral>, 1539
atomic<T*>, 1542
atomic_ref<floating-point>, 1532
atomic_ref<integral>, 1531
atomic<T*>, 1542
fetch_xor
atomic<integral>, 1539
atomic_ref<integral>, 1531
fetestexcept, 1141
feupdateenv, 1141
fexcept_t, 1141
fflush, 1482
fgetc, 1482
fgetpos, 1482
fgets, 1482
fgetwc, 781
fgetws, 781
FILE, 1482
file_clock, 1257
now, 1257
file_size, 1476
directory_entry, 1465
file_status, 1461
constructor, 1461
permissions, 1461, 1462
type, 1461
file_time, 1225
from_stream, 1258
operator<<, 1257
file_type, 1458
filebuf, 1356, 1421
filename
path, 1453
FILENAME_MAX, 1482
filesystem_error, 1457
constructor, 1457, 1458
path1, 1458
path2, 1458
what, 1458
fill, 1086
array, 823
basic_ios, 1371
fill_n, 1086
filter_view, 989
base, 989
begin, 990
constructor, 990
end, 989
iterator, 990
pred, 990
sentinel, 992
filter_view::iterator
base, 991
constructor, 991
iter_move, 992
iter_swap, 992
operator*, 991
operator++, 991
operator->, 991
operator--, 991
operator==, 992
filter_view::sentinel
base, 992
constructor, 992
operator==, 992
find, 1069
basic_string, 763
basic_string_view, 777
ordered associative containers, 805
unordered associative containers, 815
find_end, 1070
find_first_not_of
basic_string, 763
basic_string_view, 777
find_first_of, 1071
basic_string, 763
basic_string_view, 777
find_if, 1069
find_if_not, 1069
find_last_not_of
basic_string, 763
basic_string_view, 778
find_last_of
basic_string, 763
basic_string_view, 777
first
local_info, 1298
span, 898
first_argument_type
zombie, 482
fisher_distribution
result_type, 1184
fisher_f_distribution, 1184
constructor, 1184
m, 1185
n, 1185
fixed, 1374
chars_format, 718
flag_type
basic_regex, 1501
flags
ios_base, 1322, 1365
flip
bitset, 612
vector<bool>, 846
float_denorm_style, 494, 495
numeric_limits, 498
float_round_style, 494, 495
Index of library names

FP_SUBNORMAL, 1209
FP_ZERO, 1209
fpclassify, 1209
fpos, 1356, 1360, 1367, 1368
  state, 1368
fpos_t, 1482
fprintf, 1482
fputc, 1482
fputs, 1482
fputwc, 781
fputws, 781
frac_digits
  moneypunct, 1351
fread, 1482
free, 489, 630
freeze
  ostrstream, 1671
  stringstream, 1672
  strstreambuf, 1667
freopen, 1482
frexp, 1209
frexpf, 1209
frexpl, 1209
from_address
  coroutine_handle, 529
from_bytes
  wstring_convert, 1679
from_chars, 720, 721
from_chars_result, 719
  ec, 719
  ptr, 719
from_promise
  coroutine_handle, 529
from_stream
  day, 1263
  duration, 1247
  file_time, 1258
  gps_time, 1257
  local_time, 1259
  month, 1265
  month_day, 1273
  sys_time, 1252
  tai_time, 1255
  utc_time, 1253
  weekday, 1270
  year, 1267
  year_month, 1277
  year_month_day, 1280
from_sys
  utc_clock, 1253
from_time_t
  system_clock, 1251
from_utc
  gps_clock, 1256
  tai_clock, 1255
front
  basic_string, 756
  basic_string_view, 775
  forward_list, 831
  span, 899
  tzdb_list, 1294
  view_interface, 973
front_insert_iterator, 937
  constructor, 937
  operator*, 937
  operator++, 937
  operator=, 937
front_insiter, 937
fscanf, 1482
fseek, 1424, 1482
fsetpos, 1482
fstream, 1356, 1421
ftell, 1482
function, 682
  constructor, 683, 684
  destructor, 685
  invocation, 685
  operator bool, 685
  operator(), 685
  operator=, 684, 685
  operator==, 685
  result_type, 682
  swap, 685
  target, 685
  target_type, 685
future, 1601
  constructor, 1601
  get, 1602
  operator=, 1602
  share, 1602
  valid, 1602
  wait, 1602
  wait_for, 1602
  wait_until, 1602
future_category, 1597
future_errc, 1596
  make_error_code, 1597
  make_error_condition, 1597
future_error, 1597
  code, 1597
  constructor, 1597
  what, 1597
fwide, 781
fwprintf, 781
fwrite, 1482
fwscanf, 781
G
  gamma_distribution, 1179
    alpha, 1179
    beta, 1179
    constructor, 1179
    result_type, 1179
gbump
  basic_streambuf, 1379
gcd, 1133
gcount
basic_istream, 1390
get
array, 824
basic_format_args, 738
basic_istream, 1390, 1391
future, 1602
messages, 1352
money_get, 1347
num_get, 1333
pair, 570, 571
reference_wrapper, 670
shared_future, 1604
shared_ptr, 644
subrange, 977
time_get, 1343
tuple, 579
unique_ptr, 635
variant, 601
get_allocator
basic_string, 763
basic_stringbuf, 1411
basic_syncbuf, 1435
match_results, 1507
get_date
time_get, 1343
get_default_resource, 658
get_deleter
shared_ptr, 649
unique_ptr, 635
get_future
packaged_task, 1608
promise, 1599
get_id
jthread, 1564
this_thread, 1565
thread, 1562
get_if, 601
variant, 601
get_info
time_zone, 1299
zoned_time, 1303
get_leap_second_info, 1254
get_local_time
zoned_time, 1303
get_money, 1404
get_monthname
time_get, 1343
get_new_handler, 483, 510
get_pointerafety, 624
get_stop_source
jthread, 1564
generate_canonical
jthread, 1564
generate
jthread, 1564
generate_n
jthread, 1564
genetic_category, 554, 555
generic_string
path, 1452, 1453
genetic_u16string
path, 1453
genetic_u32string
path, 1453
genetic_u8string
path, 1453
genetic_wstring
path, 1453
generate
jthread, 1564
generative_distribution, 1176
constructor, 1177
p, 1177
result_type, 1176
generic
future, 1602
messages, 1352
money_get, 1347
num_get, 1333
pair, 570, 571
reference_wrapper, 670
shared_future, 1604
shared_ptr, 644
subrange, 977
time_get, 1343
tuple, 579
unique_ptr, 635
variant, 601
get_allocator
basic_string, 763
basic_stringbuf, 1411
basic_syncbuf, 1435
match_results, 1507
get_date
time_get, 1343
get_default_resource, 658
get_deleter
shared_ptr, 649
unique_ptr, 635
get_future
packaged_task, 1608
promise, 1599
get_id
jthread, 1564
this_thread, 1565
thread, 1562
get_if, 601
variant, 601
get_info
time_zone, 1299
zoned_time, 1303
get_leap_second_info, 1254
get_local_time
zoned_time, 1303
get_money, 1404
get_monthname
time_get, 1343
get_new_handler, 483, 510
get_pointerafety, 624
get_stop_source
jthread, 1564
generate_canonical
jthread, 1564
generate
jthread, 1564
generate_n
jthread, 1564
genetic_category, 554, 555
genetic_string
path, 1452, 1453
genetic_u16string
path, 1453
genetic_u32string
path, 1453
genetic_u8string
path, 1453
genetic_wstring
path, 1453
genetic
future, 1602
messages, 1352
money_get, 1347
num_get, 1333
pair, 570, 571
reference_wrapper, 670
shared_future, 1604
shared_ptr, 644
subrange, 977
time_get, 1343
tuple, 579
unique_ptr, 635
variant, 601
get_allocator
basic_string, 763
basic_stringbuf, 1411
basic_syncbuf, 1435
match_results, 1507
get_date
time_get, 1343
get_default_resource, 658
get_deleter
shared_ptr, 649
unique_ptr, 635
get_future
packaged_task, 1608
promise, 1599
get_id
jthread, 1564
this_thread, 1565
thread, 1562
get_if, 601
variant, 601
get_info
time_zone, 1299
zoned_time, 1303
get_leap_second_info, 1254
get_local_time
zoned_time, 1303
get_money, 1404
get_monthname
time_get, 1343
get_new_handler, 483, 510
get_pointerafety, 624
get_stop_source
jthread, 1564
generate_canonical
jthread, 1564
generate
jthread, 1564
generate_n
jthread, 1564

Index of library names
now, 1256
to_utc, 1256
gps_seconds, 1225
gps_time, 1225
    from_stream, 1257
operator<<, 1256
gptr
    basic_streambuf, 1379
greater, 674, 676
    operator(), 674
    partial_ordering, 521
    strong_ordering, 523
    weak_ordering, 522
greater<>, 674
    operator(), 674
greater_equal, 674, 676
    operator(), 674
greater_equal<>, 674
    operator(), 674
grep
    syntax_option_type, 1492, 1493
grouping
    moneypunct, 1351
    numpunct, 1339
gslice, 1204
    constructor, 1205
    size, 1205
    start, 1205
    stride, 1205
gslice_array, 1205
    operator=, 1206
    operator+=, 1206
    operator-=, 1206
    operator/, 1206
    operator<<=, 1206
    operator>=, 1206
    operator%=, 1206
    operator&=, 1206
    operator|=, 1206
    operator!?, 1206
    value_type, 1205
H
handle
    basic_format_arg, 737
hard_link_count, 1476
directory_entry, 1465
hardware_concurrency
    jthread, 1564
    thread, 1562
hardware_constructive_interference_size, 511
hardware_destructive_interference_size, 511
has_denorm_loss
    numeric_limits, 498
has_extension
    path, 1455
has_facet
    locale, 1322
has_filename
    path, 1455
has_infinity
    numeric_limits, 498
has_parent_path
    path, 1455
has_quiet_NaN
    numeric_limits, 498
has_relative_path
    path, 1455
has_root_directory
    path, 1454
has_root_name
    path, 1454
has_root_path
    path, 1454
has_signaling_NaN
    numeric_limits, 498
has_single_bit, 1151
has_stem
    path, 1455
has_unique_object_representations, 702, 703
has_value
    any, 607
    optional, 589
has_virtualDestructor, 702
has_virtualDestructor_v, 695
hash, 688
    collate, 1341
    coroutine_handle, 530
    error_code, 559
    monostate, 604
    optional, 593
    pmr::string, 770
    pmr::u16string, 770
    pmr::u32string, 770
    pmr::wstring, 770
    shared_ptr, 653
    string, 770
    string_view, 779
    thread::id, 1560
    type_index, 717
    u16string, 770
    u16string_view, 779
    u32string, 770
    u32string_view, 779
    u8string_view, 779
    unique_ptr, 653
    variant, 604
    wstring, 770
    wstring_view, 779
hash_code
    unordered associative containers, 811

Index of library names
hash_value
path, 1457
hasher
unordered associative containers, 809
hermite, 1222
hermitef, 1222
hermitel, 1222
hex, 1374
chars_format, 718
hexfloat, 1374
hh_mm_ss
hours, 1292
is_negative, 1292
minutes, 1292
operator precision, 1292
seconds, 1292
subseconds, 1292
to_duration, 1292
high_resolution_clock, 1258
hms, 1290
holds_alternative, 601
variant, 601
hours, 1225
hh_mm_ss, 1292
HUGE_VAL, 1209
HUGE_VALF, 1209
HUGE_VALL, 1209
hypot, 1209
3-argument form, 1218
hypotf, 1209
hypotl, 1209
I
icase
syntax_option_type, 1492, 1493
id
locale, 1320
thread, 1559
identity, 679
ifstream, 1356, 1421
ignore, 577
basic_istream, 1392
ilogb, 1209
ilogbf, 1209
ilogbl, 1209
imag, 1150
complex, 1146, 1148
imaxabs, 1483
imaxdiv, 1483
imake, 1483
imake_t, 1483
imbue, 1497
basic_filebuf, 1427
basic_ios, 1370
basic_regex, 1501
basic_filebuf, 1380
ios_base, 1366
in
codecvt, 1329
in_avail
basic_streambuf, 1378
in_place, 563, 615
in_place_index, 563
in_place_index_t, 563
in_place_t, 563
in_place_type, 563
in_place_type_t, 563
in_range, 566
includes, 1107
inclusive_scan, 1129
increment
directory_iterator, 1467
recursive_directory_iterator, 1469
incrementable, 917
incrementable_traits, 909
independent_bits_engine, 1167, 1168
result_type, 1168
index
variant, 600
weekday_indexed, 1271
year_month_weekday, 1284
index_sequence, 562
index_sequence_for, 562
indirect_array, 1207
operator**, 1208
operator**, 1208
operator**, 1208
operator/, 1208
operator<<, 1208
operator*=, 1208
operator/=, 1208
operator%=, 1208
operator&=, 1208
operator^=, 1208
operator|=, 1208
value_type, 1207
indirect_binary_predicate, 925
indirect_equivalence_relation, 925
indirect_strict_weak_order, 926
indirect Unary predicate, 925
indirectly_comparable, 927
indirectly_Copyable, 927
indirectly_Copyable, 927
indirectly_Movable, 926
indirectly_Movable, 926
indirectlyReadable, 915
indirectlyReadable_traits, 910
indirectlyRegular Unary Invocable, 925
indirectlySwappable, 927
indirectlyUnaryInvocable, 925
indirectlyWritable, 915
infinity
numeric_limits, 498
Init
ios_base, 1365
init
basic_ios, 1370, 1386
basic_ostream, 1397
initializer_list, 519
begin, 519
constructor, 519
end, 519
size, 519
inner_allocator
  scoped_allocator_adaptor, 665
inner_allocator_type
  scoped_allocator_adaptor, 664
inner_product, 1126
inplace_merge, 1106
input_iterator, 918
input_iterator_tag, 928
input_or_output_iterator, 917
input_range, 971
insert
  basic_string, 758–760
ddeque, 827
list, 837
map, 852
multimap, 857
ordered associative containers, 801
unordered associative containers, 812
unordered_map, 871
unordered_multimap, 877
vector, 843
insert_after
  forward_list, 831
insert_iterator, 937
  constructor, 938
  operator*, 938
  operator++, 938
  operator=, 938
insert_or_assign
  map, 853
  unordered_map, 872
inserter, 938
int16_t, 502
int32_t, 502
int64_t, 502
int8_t, 502
int_fast16_t, 502
int_fast32_t, 502
int_fast64_t, 502
int_fast8_t, 502
int_least16_t, 502
int_least32_t, 502
int_least64_t, 502
int_least8_t, 502
INT_MAX, 501
INT_MIN, 501
int_type
  char_traits, 741
  wstring_convert, 1679
integer_sequence, 566
  value_type, 566
integral, 539
integral_constant, 696
  value_type, 696
internal, 1374
intervals
  piecewise_constant_distribution, 1188
  piecewise_linear_distribution, 1189
intmax_t, 502
intptr_t, 502
invalid_argument, 547, 548, 611
  constructor, 548
invocable, 545
INVOKE, 668, 669
invoke, 669
invoke_result_t, 692
io_errc, 1359
  make_error_code, 1375
  make_error_condition, 1375
io_state
  zombie, 482
ios, 1356, 1360
ios_base, 1361
  constructor, 1367
destructor, 1367
failure, 1363
flags, 1322, 1365
fmtflags, 1363, 1403
getloc, 1366
imbue, 1366
Init, 1365
iostate, 1364
iword, 1366
openmode, 1364
precision, 1322, 1365
pword, 1367
register_callback, 1367
seekdir, 1364
setf, 1365
sync_with_stdio, 1366
unsetf, 1365
width, 1322, 1365, 1366
xalloc, 1366
ios_base::failure, 1363
  constructor, 1363
ios_base::Init, 1365
  constructor, 1365
destructor, 1365
iostate
  iostate, 1364
iostream_category, 1375
iota, 1133
iota_view, 979
  begin, 981
  constructor, 981
  end, 981
  size, 981
iota_view::iterator, 981
  constructor, 982
  operator*, 983
  operator+, 984
is##am 1293
is##arithmetic, 697
is##arithmetic_v, 693
is##array, 696
is##array_v, 693
is##assignable, 699
is##assignable_v, 694
is##base_of, 704
is##base_of_v, 695
is##bind_expression, 680
is##bind_expression_v, 667
is##block_file, 1476
directory_entry, 1464
is##bounded
numeric_limits, 499
is##bounded_array, 698
is##bounded_array_v, 694
is##character_file, 1476
directory_entry, 1464
is##class, 697
is##class_v, 693
is##clock, 1240
is##clock_v, 1225
is##compound, 697
is##compound_v, 693
is##const, 698
is##const_v, 693
is##constructible, 698, 703
is##constructible_v, 694
is##convertible, 704, 705
is##convertible_v, 695
is##copy_assignable, 699
is##copy_assignable_v, 694
is##copy_constructible, 699
is##copy_constructible_v, 694
is##corresponding member, 712
is##default_constructible, 699
is##default_constructible_v, 694
is##destructible, 700
is##destructible_v, 694
is##directory, 1476
directory_entry, 1464
is##empty
class, 698
function, 1477
is##empty_v, 694
is##enum, 697
is##enum_v, 693
is##eq, 520
is##equal
memory_resource, 654
is##error_code_enum, 552
is##error_condition_enum, 552
is##exact
numeric_limits, 497
is##execution_policy, 717
is##execution_policy_v, 717
is##fifo, 1477
directory_entry, 1464
is##final, 698
is##final_v, 694
is##floating_point, 696
is##floating_point_v, 693
is##function, 697
is##function_v, 693
is##fundamental, 697
is##fundamental_v, 693
is##geq, 520
is##gt, 520
is##gteq, 520
is##heap, 1114
is##heap_until, 1114
is##iec559
numeric_limits, 499
is##integer
numeric_limits, 497
is##integral, 696
is##integral_v, 693
is##invocable, 705
is_trivially_copy_constructible, 701
is_trivially_copyable, 698
is_trivially_copyable_v, 694
is_trivially_default_constructible, 701
is_trivially_destructible, 701
is_trivially_destructible_v, 695
is_trivially_move_assignable, 701
is_trivially_move_constructible, 701
is_unbounded_array, 698
is_unbounded_array_v, 694
is_trivial, 697
is_union, 697
is_union_v, 693
is_unsigned, 698
is_unsigned_v, 694
is_void, 696
is_void_v, 693
is_volatile, 696
is_volatile_v, 693
isalnum, 779, 1322
isalpha, 779, 1322
isblank, 779, 1322
iscntrl, 779, 1322
isctype
regex_traits, 1496
regular expression traits, 1518
isdigit, 779, 1322
isfinite, 1209
isgraph, 779, 1322
isgreater, 1209
isgreaterequal, 1209
isinf, 1209
isless, 1209
islessequal, 1209
islessgreater, 1209
islower, 779, 1322
isnormal, 1209
isunordered, 1209
iswalnum, 780
iswalpha, 780
iswblank, 780
iswcntrl, 780
iswctype, 780
iswdigit, 780
iswgraph, 780
iswlower, 780
iswprint, 780
iswpunct, 780
iswspace, 780
iswupper, 780
iswxdigit, 780
isxdigit, 779, 1322
iter_difference_t, 910
iter_move, 913
custom_iterator, 947
counted_iterator, 951
filter_view::iterator, 992
move_iterator, 942
reverse_iterator, 935
iter_swap, 914, 1082
custom_iterator, 947
counted_iterator, 951
filter_view::iterator, 992
join_view::iterator, 1009
move_iterator, 942
reverse_iterator, 935
split_view::inner_iterator, 1014
transform_view::iterator, 997
iter_value_t, 911
iterator, 1673
basic_format_context, 733
basic_format_parse_context, 732
basic_string, 747
basic_string_view, 771
filter_view, 990
path, 1456
span, 899
transform_view::iterator, 996
iterator_category
custom_traits, 911
iterator_traits, 911
iterator_category, 911
pointer, 911
reference, 911
iword
ios_base, 1366
J
jmp_buf, 532
join
Index of library names

jthread, 1564
thread, 1561
join_view, 1005
  base, 1005
  begin, 1005
  constructor, 1006
  end, 1005
join_view::iterator, 1006
  constructor, 1008
  iter_swap, 1009
  operator++, 1008
  operator->, 1008
  operator--, 1008
  operator==, 1009
join_view::sentinel, 1009
  constructor, 1009
  operator==, 1009
joinable
  jthread, 1564
  thread, 1561
jthread, 1562
  constructor, 1563
  detach, 1564
  get_id, 1564
  get_stop_source, 1564
  get_stop_token, 1564
  hardware_concurrency, 1564
  join, 1564
  joinable, 1564
  operator=, 1563
  request_stop, 1564
  swap, 1564

K
k  negative_binomial_distribution, 1177
key_comp
  ordered associative containers, 800
key_compare
  ordered associative containers, 799
key_eq
  unordered associative containers, 811
key_equal
  unordered associative containers, 809
key_type
  ordered associative containers, 799
  unordered associative containers, 808
kill_dependency, 1526
knuth_b, 1170

L
L_tmpnam, 1482
labs, 489
laguerre, 1222
laguerref, 1222
laguerrel, 1222
lambda
exponential_distribution, 1179
largest_required_pool_block
  pool_options, 659
last
  span, 898, 899
last Spec, 1262
last_write_time, 1478
  directory_entry, 1465
latch
  arrive_and_wait, 1593
  constructor, 1593
  count_down, 1593
  max, 1593
  try_wait, 1593
  wait, 1593
latest
  choose, 1225
launder, 510
LC_ALL, 1353
LC_COLLATE, 1353
LC_CTYPE, 1353
LC_MONETARY, 1353
LC_NUMERIC, 1353
LC_TIME, 1353
1cm, 1133
l_conv, 1353
LDBL_DECIMAL_DIG, 502
LDBL_DIG, 502
LDBL_EPSILON, 502
LDBL_HAS_SUBNORM, 502
LDBL_MANT_DIG, 502
LDBL_MAX, 502
LDBL_MAX_10_EXP, 502
LDBL_MAX_EXP, 502
LDBL_MIN, 502
LDBL_MIN_10_EXP, 502
LDBL_MIN_EXP, 502
LDBL_TRUE_MIN, 502
ldexp, 1209
ldexpf, 1209
ldexpl, 1209
ldiv, 489
ldiv_t, 489
leap second, 1304
date, 1304
operator<, 1305
operator<=, 1305
operator<==, 1305
operator==, 1305
operator>, 1305
operator>=, 1305
value, 1304
leap_second info, 1254
left, 1374
legendre, 1222
legendref, 1222
legendrel, 1222
length
  basic_string, 755
Index of library names

atomic, 1535
atomic<floating-point>, 1535
atomic<integral>, 1535
atomic<shared_ptr<T>>, 1545
atomic<T*>, 1535
atomic<weak_ptr<T*>>, 1547
atomic_ref, 1528
atomic_ref-floating-point, 1528
atomic_ref-integral, 1528
atomic_ref<T*>, 1528
load_factor
unordered associative containers, 817
local-time-format-t, 1309
local_days, 1225
local_info, 1298
ambiguous, 1298
first, 1298
nonexistent, 1298
operator<<, 1298
result, 1298
second, 1298
unique, 1298
local_iterator
unordered associative containers, 809
local_seconds, 1225
local_t, 1225
local_time, 1225, 1258
from_stream, 1259
operator<<, 1258
local_time_format, 1309
locale, 1497, 1501, 1517
basic_format_context, 734
category, 1318
classic, 1322
combine, 1321
constructor, 1320, 1321
facet, 1319
global, 1321
has_facet, 1322
id, 1320
name, 1321
operator(), 1321
operator=, 1321
operator==, 1321
use_facet, 1322
localeconv, 1353
localtime, 1313
locate_zone, 1295
tzdb, 1293
zoned_traits< const time_zone*>, 1300
lock, 1582
shared_lock, 1580
unique_lock, 1577
weak_ptr, 651
lock_guard, 1573
constructor, 1574
destructor, 1574
log, 1209
complex, 1149
valarray, 1201
log10, 1209
  complex, 1149
  valarray, 1201
log10f, 1209
log10l, 1209
log1pf, 1209
log1pl, 1209
log2, 1209
log2f, 1209
log2l, 1209
logb, 1209
logbf, 1209
logbl, 1209
logf, 1209
logic_error, 547
  constructor, 547, 548
logical_and, 677
  operator(), 677
logical_and<>, 677
  operator(), 677
logical_not, 677
  operator(), 677
logical_not<>, 678
  operator(), 678
logical_or, 677
  operator(), 677
logical_or<>, 677
  operator(), 677
log1l, 1209
lognormal_distribution, 1182
  constructor, 1182
  m, 1182
  result_type, 1182
  s, 1182
LONG_MAX, 501
LONG_MIN, 501
longjmp, 532
lookup_classname
  regex_traits, 1496
    regular expression traits, 1518
lookup_collatename
  regex_traits, 1496
    regular expression traits, 1518
lower_bound, 1100
  ordered associative containers, 806
lowest
  numeric_limits, 497
lrint, 1209
lrintf, 1209
lrintl, 1209
lround, 1209
lroundf, 1209
lroundl, 1209

M

  fisher_f_distribution, 1185
  lognormal_distribution, 1182
make12, 1293
make24, 1293
make_any, 608
make_error_code
  errc, 557
    future_errc, 1597
  io_errc, 1375
make_error_condition
  errc, 558
    future_errc, 1597
  io_errc, 1375
make_exception_ptr, 518
make_format_args, 737
make_from_tuple, 578
make_heap, 1113
make_index_sequence, 562
make_integer_sequence, 567
make_move_iterator, 942
make_obj_using_allocator, 627
make_optional, 592, 593
make_pair, 570
make_preferred
  path, 1451
make_ready_at_thread_exit
  packaged_task, 1608
make_reverse_iterator, 935
make_shared, 489, 630, 1660
map, 849
  at, 852
    clear, 805
      constructor, 799, 852
    contains, 805
    count, 805
    emplace, 801
    emplace_hint, 801
    equal_range, 806
    erase, 805
    erase_if, 854
    extract, 804
    find, 805
    insert, 801, 852
    insert_or_assign, 853
    key_comp, 800
    key_compare, 799
    key_type, 799

Index of library names
lower_bound, 806
mapped_type, 799
merge, 804
node_type, 799
operator<, 852
operator==, 852
try_emplace, 853
upper_bound, 806
value_comp, 800
value_compare, 799
value_type, 799
map::value_compare
comp, 849
operator(), 849
mapped_type
ordered associative containers, 799
unordered associative containers, 809
mark_count
basic_regex, 1501
mask_array, 1206
operator==, 1207
operator+, 1207
operator-, 1207
operator-, 1207
operator<<=, 1207
operator=, 1207
operator>=, 1207
operator[][], 1207
operator[][], 1207
operator[], 1207
operator[][], 1207
operator()[, 1206
operator%?, 1207
operator%=, 1207
operator*=, 1207
operator%=, 1207
operator[]=, 1207
value_type, 1206
match_any, 1492, 1494
match_continuous, 1492, 1494, 1513
match_default, 1492
match_flag_type, 1492, 1493, 1518
regex_constants, 1492
match_not_bol, 1492, 1493
match_not_bow, 1492, 1494
match_not_eol, 1492, 1494
match_not_eow, 1492, 1494
match_not_null, 1492, 1494, 1513
match_prev_avail, 1492, 1494, 1513
match_results, 1503, 1512, 1514
begin, 1506
constructor, 1505
empty, 1506
end, 1506
format, 1506, 1507
get_allocator, 1507
length, 1506
matched, 1504
max_size, 1506
operator=, 1505
operator==, 1507
operator[], 1506
position, 1506
prefix, 1506
ready, 1505
size, 1505
str, 1506
suffix, 1506
swap, 1507
matched
match_results, 1504
MATH_ERREXCEPT, 1209
math_errhandling, 1209
MATH_ERRNO, 1209
max, 1115, 1116
barrier, 1594
counting_semaphore, 1591
duration, 1243
duration_values, 1240
latch, 1593
numeric_limits, 496
time_point, 1249
valarray, 1199
year, 1267
max_align_t, 488, 491
max_blocks_per_chunk
pool_options, 659
max_bucket_count
unordered associative containers, 816
max_digits10
numeric_limits, 497
max_element, 1117
max_exponent
numeric_limits, 498
max_exponent10
numeric_limits, 498
max_length
codecvt, 1329
max_load_factor
unordered associative containers, 817
max_size
allocator_traits, 629
array, 821
basic_string, 755
basic_string_view, 774
match_results, 1506
scoped_allocator_adaptor, 665
MB_CUR_MAX, 489
MB_LEN_MAX, 501
mblen, 489, 783
mbrlen, 781, 783
mbrstowcs, 783
mbtowc16, 782
mbtowc32, 782
mbtowc8, 782, 783
mbstowcs, 781, 783
mbstate_t, 781, 782
mbstowcs, 489, 783
mbtowc, 489, 783
mean
normal_distribution, 1182
Index of library names

negative_sign, 1351
positive_sign, 1351
thousands_sep, 1351
moneypunct_byname, 1352
monostate, 603
hash, 604
operator<>, 603
operator=, 603
monotonic_buffer_resource, 661
constructor, 661
destructor, 661
do_allocate, 662
do_deallocate, 662
do_is_equal, 662
release, 662
upstream_resource, 662
month, 1264
constructor, 1264
from_stream, 1265
month_day, 1272
month_day_last, 1273
month_weekday, 1274
month_weekday_last, 1275
ok, 1265
operator unsigned, 1265
operator+, 1265
operator++, 1264
operator=+, 1264
operator-, 1265
operator=-, 1264
operator=, 1264
operator<, 1265
operator<>, 1265
operator==, 1265
year_month, 1276
year_month_day, 1279
year_month_day_last, 1281
year_month_weekday, 1284
year_month_weekday_last, 1286
month_day, 1272
constructor, 1272
day, 1272
from_stream, 1273
month, 1272
ok, 1272
operator<<, 1273
operator>=, 1273
operator==, 1273
year_month_day, 1279
year_month_day_last, 1281
year_month_weekday, 1284
year_month_weekday_last, 1286
month_day_last, 1273
constructor, 1273
month, 1273
ok, 1273
operator<<, 1274
operator>=, 1274
operator==, 1273
year_month_day_last, 1282
month_weekday, 1274
constructor, 1274
month, 1274
ok, 1274
operator<<, 1274
operator==, 1274
weekday_indexed, 1274
month_weekday_last, 1275
constructor, 1275
month, 1275
ok, 1275
operator<<, 1275
operator==, 1275
weekday_last, 1275
months, 1225
movable, 545
move
algorithm, 1080, 1081
basic_ios, 1371
function, 564
move_backward, 1081
move_constructible, 542
move_if_noexcept, 565
move_iterator, 939
base, 940
constructor, 940
iter_move, 942
iter_swap, 942
operator*, 940
operator*, 941, 942
operator++, 940
operator++, 941
operator=+, 941
operator=, 941, 942
operator=, 941
operator->, 1674
operator->, 941
operator<, 941
operator<, 941
operator<, 942
operator=, 940
operator=, 941
operator=, 941
operator=, 941
operator[], 940
move_sentinel, 942
base, 943
constructor, 943
operator=, 943
mt19937, 1169
mt19937_64, 1169
multiline
syntax_option_type, 1493
multimap, 854
clear, 805
constructor, 799, 857
contains, 805
count, 805
emplace, 801
emplace_hint, 801
equal_range, 806
erase, 805
erase_if, 857
extract, 804
find, 805
insert, 801, 857
key_comp, 800
key_compare, 799
key_type, 799
lower_bound, 806
mapped_type, 799
merge, 804
node_type, 799
operator<, 857
operator==, 857
upper_bound, 806
value_comp, 800
value_compare, 799
value_type, 799
multimap::value_compare
comp, 854
operator(), 854
multiplies, 672
operator(), 672
multiplies<>, 672
operator(), 672
multiset, 861
clear, 805
constructor, 799, 864
contains, 805
count, 805
eemplace, 801
eemplace_hint, 801
equal_range, 806
erase, 805
erase_if, 864
extract, 804
find, 805
insert, 801
key_comp, 800
key_compare, 799
key_type, 799
lower_bound, 806
mapped_type, 799
merge, 804
node_type, 799
operator<, 864
operator==, 864
upper_bound, 806
value_comp, 800
value_compare, 799
value_type, 799
mutex, 1567
shared_lock, 1581
unique_lock, 1578

N
n
chi_squared_distribution, 1183
fisher_f_distribution, 1185
name

Index of library names 1786
Index of library names

- **atomic<weak_ptr<T>>**, 1548
- **atomic_ref<T>**, 1529
- **condition_variable**, 1585
- **condition_variable_any**, 1588
- **nounitbuf**, 1373
- **nouppercase**, 1373
- **now**
  - **file_clock**, 1257
  - **gps_clock**, 1256
  - **tai_clock**, 1254
  - **utc_clock**, 1252
- **nth_element**, 1099
- **NULL**, 488–490, 781, 1313, 1353, 1482
- **null_memory_resource**, 657
- **nullopt**, 590
- **nullptr_t**, 590
- **nullptr_t**, 488, 490
- **num_get**, 1332
  - **do_get**, 1333, 1335
  - **get**, 1333
- **num_put**, 1335
  - **do_put**, 1336, 1338
  - **put**, 1336
- **numeric_limits**, 494, 495
  - **denorm_min**, 499
  - **digits**, 497
  - **digits10**, 497
  - **epsilon**, 497
  - **float_denorm_style**, 498
  - **has_denorm_loss**, 498
  - **has_infinity**, 498
  - **has_quiet_NaN**, 498
  - **has_signaling_NaN**, 498
  - **infinity**, 498
  - **is_bounded**, 499
  - **is_exact**, 497
  - **is_iec559**, 499
  - **is_integer**, 497
  - **is_modulo**, 499
  - **is_signed**, 497
  - **lowest**, 497
  - **max**, 496
  - **max_digits10**, 497
  - **max_exponent**, 498
  - **max_exponent10**, 498
  - **min**, 496
  - **min_exponent**, 497
  - **min_exponent10**, 498
  - **quiet_NaN**, 499
  - **radix**, 497
  - **round_error**, 497
  - **round_style**, 499
  - **signaling_NaN**, 499
  - **tinyness_before**, 499
  - **traps**, 499
- **numeric_limits<bool>**, 500
- **numpunct**, 1338
  - **decimal_point**, 1339
  - **do_decimal_point**, 1340
Index of library names
Index of library names
Here is the plain text representation of the document:

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostreambuf_iterator</td>
<td>957</td>
<td>1513</td>
</tr>
<tr>
<td>regex_iterator</td>
<td></td>
<td>1516</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td></td>
<td>933</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>644</td>
<td>644</td>
</tr>
<tr>
<td>split_view::outer-iterator</td>
<td>1012</td>
<td>1512</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>634</td>
<td>634</td>
</tr>
<tr>
<td>valarray</td>
<td>1199</td>
<td>1200</td>
</tr>
<tr>
<td>operator**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>1146</td>
<td>1147</td>
</tr>
<tr>
<td>duration</td>
<td>1243</td>
<td></td>
</tr>
<tr>
<td>gslice_array</td>
<td>1206</td>
<td></td>
</tr>
<tr>
<td>indirect_array</td>
<td>1208</td>
<td></td>
</tr>
<tr>
<td>mask_array</td>
<td>1207</td>
<td></td>
</tr>
<tr>
<td>slice_array</td>
<td>1203</td>
<td></td>
</tr>
<tr>
<td>valarray</td>
<td>1198</td>
<td></td>
</tr>
<tr>
<td>operator+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>basic_string</td>
<td>765</td>
<td>766</td>
</tr>
<tr>
<td>complex</td>
<td>1147</td>
<td></td>
</tr>
<tr>
<td>counted_iterator</td>
<td>950</td>
<td></td>
</tr>
<tr>
<td>day</td>
<td>1263</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>1242</td>
<td>1249</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1022</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>984</td>
<td></td>
</tr>
<tr>
<td>month</td>
<td>1265</td>
<td></td>
</tr>
<tr>
<td>move_iterator</td>
<td>941</td>
<td>942</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>933</td>
<td>935</td>
</tr>
<tr>
<td>time_point</td>
<td>1249</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>997</td>
<td></td>
</tr>
<tr>
<td>valarray</td>
<td>1197</td>
<td>1199</td>
</tr>
<tr>
<td>weekday</td>
<td>1269</td>
<td>1270</td>
</tr>
<tr>
<td>year</td>
<td>1267</td>
<td></td>
</tr>
<tr>
<td>year_month</td>
<td>1277</td>
<td></td>
</tr>
<tr>
<td>year_month_day</td>
<td>1280</td>
<td></td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1282</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday</td>
<td>1285</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday_last</td>
<td>1287</td>
<td></td>
</tr>
<tr>
<td>operator++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atomic&lt;integral&gt;</td>
<td>1543</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;T*&gt;</td>
<td>1543</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;integral&gt;</td>
<td>1533</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;T*&gt;</td>
<td>1533</td>
<td></td>
</tr>
<tr>
<td>back_insert_iterator</td>
<td>936</td>
<td></td>
</tr>
<tr>
<td>basic_istream_view::iterator</td>
<td>986</td>
<td>987</td>
</tr>
<tr>
<td>common_iterator</td>
<td>946</td>
<td></td>
</tr>
<tr>
<td>counted_iterator</td>
<td>949</td>
<td>950</td>
</tr>
<tr>
<td>day</td>
<td>1262</td>
<td></td>
</tr>
<tr>
<td>directory_iterator</td>
<td>1467</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>1242</td>
<td>1243</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1021</td>
<td></td>
</tr>
<tr>
<td>filter_view::iterator</td>
<td>991</td>
<td></td>
</tr>
<tr>
<td>front_insert_iterator</td>
<td>937</td>
<td></td>
</tr>
<tr>
<td>insert_iterator</td>
<td>938</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>983</td>
<td></td>
</tr>
<tr>
<td>istream_iterator</td>
<td>953</td>
<td></td>
</tr>
<tr>
<td>istreambuf_iterator</td>
<td>956</td>
<td></td>
</tr>
<tr>
<td>join_view::iterator</td>
<td>1008</td>
<td></td>
</tr>
<tr>
<td>month</td>
<td>1264</td>
<td></td>
</tr>
<tr>
<td>move_iterator</td>
<td>940</td>
<td></td>
</tr>
</tbody>
</table>

Index of library names

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>ostream_iterator</td>
<td>954</td>
<td></td>
</tr>
<tr>
<td>ostreambuf_iterator</td>
<td>957</td>
<td></td>
</tr>
<tr>
<td>recursive_directory_iterator</td>
<td>1469</td>
<td></td>
</tr>
<tr>
<td>regex_iterator</td>
<td>1513</td>
<td>1514</td>
</tr>
<tr>
<td>regex_token_iterator</td>
<td>1517</td>
<td></td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>933</td>
<td></td>
</tr>
<tr>
<td>split_view::inner-iterator</td>
<td>1014</td>
<td></td>
</tr>
<tr>
<td>split_view::outer-iterator</td>
<td>1012</td>
<td></td>
</tr>
<tr>
<td>time_point</td>
<td>1249</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>996</td>
<td></td>
</tr>
<tr>
<td>weekday</td>
<td>1269</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>1266</td>
<td></td>
</tr>
<tr>
<td>operator+=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>atomic&lt;floating-point&gt;</td>
<td>1541</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;integral&gt;</td>
<td>1539</td>
<td></td>
</tr>
<tr>
<td>atomic&lt;T*&gt;</td>
<td>1539</td>
<td>1541</td>
</tr>
<tr>
<td>atomic_ref&lt;floating-point&gt;</td>
<td>1532</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;integral&gt;</td>
<td>1531</td>
<td></td>
</tr>
<tr>
<td>atomic_ref&lt;T*&gt;</td>
<td>1533</td>
<td></td>
</tr>
<tr>
<td>basic_string</td>
<td>756</td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>1146</td>
<td></td>
</tr>
<tr>
<td>counted_iterator</td>
<td>950</td>
<td></td>
</tr>
<tr>
<td>day</td>
<td>1263</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>1243</td>
<td></td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1021</td>
<td></td>
</tr>
<tr>
<td>gslice_array</td>
<td>1206</td>
<td></td>
</tr>
<tr>
<td>indirect_array</td>
<td>1208</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>983</td>
<td></td>
</tr>
<tr>
<td>mask_array</td>
<td>1207</td>
<td></td>
</tr>
<tr>
<td>month</td>
<td>1264</td>
<td></td>
</tr>
<tr>
<td>move_iterator</td>
<td>941</td>
<td></td>
</tr>
<tr>
<td>path</td>
<td>1450</td>
<td>1451</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>934</td>
<td></td>
</tr>
<tr>
<td>slice_array</td>
<td>1203</td>
<td></td>
</tr>
<tr>
<td>time_point</td>
<td>1249</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>996</td>
<td></td>
</tr>
<tr>
<td>valarray</td>
<td>1198</td>
<td></td>
</tr>
<tr>
<td>weekday</td>
<td>1269</td>
<td></td>
</tr>
<tr>
<td>year</td>
<td>1266</td>
<td></td>
</tr>
<tr>
<td>year_month</td>
<td>1276</td>
<td></td>
</tr>
<tr>
<td>year_month_day</td>
<td>1278</td>
<td>1279</td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1281</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday</td>
<td>1284</td>
<td></td>
</tr>
<tr>
<td>year_month_weekday_last</td>
<td>1286</td>
<td></td>
</tr>
<tr>
<td>operator-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>common_iterator</td>
<td>946</td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td>1147</td>
<td></td>
</tr>
<tr>
<td>counted_iterator</td>
<td>950</td>
<td>951</td>
</tr>
<tr>
<td>day</td>
<td>1263</td>
<td></td>
</tr>
<tr>
<td>duration</td>
<td>1242</td>
<td>1249</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1022</td>
<td></td>
</tr>
<tr>
<td>elements_view::sentinel</td>
<td>1023</td>
<td></td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>984</td>
<td></td>
</tr>
<tr>
<td>month</td>
<td>1265</td>
<td></td>
</tr>
<tr>
<td>move_iterator</td>
<td>941</td>
<td>942</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>933</td>
<td>935</td>
</tr>
<tr>
<td>time_point</td>
<td>1249</td>
<td></td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>997</td>
<td></td>
</tr>
<tr>
<td>transform_view::sentinel</td>
<td>998</td>
<td></td>
</tr>
</tbody>
</table>

N4861
valarray, 1197, 1199, 1200
weekday, 1270
year, 1267
year_month, 1277
year_month_day, 1280
year_month_day_last, 1282
year_month_weekday, 1285
year_month_weekday_last, 1287
operator--
day, 1262
elements_view::iterator, 1021
month, 1264
time_point, 1249
weekday, 1269
year, 1266
operator= 
atomic<floating-point>, 1541
atomic<integral>, 1539
atomic<T*>, 1539, 1541, 1543
atomic_ref<floating-point>, 1532
atomic_ref<integral>, 1531
atomic_ref<T*>, 1533
complex, 1146, 1147
counted_iterator, 951
day, 1263
duration, 1243
elements_view::iterator, 1021
gslice_array, 1206
indirect_array, 1208
iota_view::iterator, 983
mask_array, 1207
month, 1264
move_iterator, 941
reverse_iterator, 934
slice_array, 1203
time_point, 1249
transform_view::iterator, 996
valarray, 1198
weekday, 1269
year, 1267
year_month, 1276
year_month_day, 1279
year_month_day_last, 1281
year_month_weekday, 1284
year_month_weekday_last, 1286
operator->
common_iterator, 945
filter_view::iterator, 991
istream_iterator, 953
iota_view::iterator, 1008
move_iterator, 1674
optional, 589
regex_iterator, 1513
regex_token_iterator, 1516
reverse_iterator, 933
shared_ptr, 644
unique_ptr, 635
operator--
atomic<integral>, 1543
atomic<T*>, 1543
atomic_ref<integral>, 1533
atomic_ref<T*>, 1533
duration, 1243
filter_view::iterator, 991
iota_view::iterator, 983
join_view::iterator, 1008
move_iterator, 941
reverse_iterator, 933, 934
transform_view::iterator, 996
operator/
calendar types, 1287–1290
complex, 1147
duration, 1244
path, 1457
valarray, 1199, 1200
operator/= 
complex, 1146, 1147
duration, 1243
gslice_array, 1206
indirect_array, 1208
mask_array, 1207
path, 1450
slice_array, 1203
valarray, 1198
duration, 1244
elements_view::iterator, 1021
iota_view::iterator, 983
mask_array, 1207
month, 1264
move_iterator, 941
reverse_iterator, 934
slice_array, 1203
time_point, 1249
transform_view::iterator, 996
valarray, 1198
weekday, 1269
year, 1267
year_month, 1276
year_month_day, 1279
year_month_day_last, 1281
year_month_weekday, 1284
year_month_weekday_last, 1286
operator->
common_iterator, 945
filter_view::iterator, 991
istream_iterator, 953
iota_view::iterator, 1008
move_iterator, 1674
optional, 589
regex_iterator, 1513
regex_token_iterator, 1516
reverse_iterator, 933
shared_ptr, 644
unique_ptr, 635
operator--
atomic<integral>, 1543
atomic<T*>, 1543
atomic_ref<integral>, 1533
atomic_ref<T*>, 1533
duration, 1243
filter_view::iterator, 991
iota_view::iterator, 983
join_view::iterator, 1008
move_iterator, 941
reverse_iterator, 933, 934
transform_view::iterator, 996
operator<<
basic_ostream, 1398, 1400–1402
basic_string, 767
basic_string_view, 779
bitset, 613, 614
byte, 491
complex, 1148
day, 1263
duration, 1247

Index of library names
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_code</td>
<td>557</td>
</tr>
<tr>
<td>file_time</td>
<td>1257</td>
</tr>
<tr>
<td>gps_time</td>
<td>1256</td>
</tr>
<tr>
<td>local_info</td>
<td>1298</td>
</tr>
<tr>
<td>local_time</td>
<td>1258</td>
</tr>
<tr>
<td>month</td>
<td>1265</td>
</tr>
<tr>
<td>month_day</td>
<td>1273</td>
</tr>
<tr>
<td>month_day_last</td>
<td>1274</td>
</tr>
<tr>
<td>month_weekday</td>
<td>1274</td>
</tr>
<tr>
<td>month_weekday_last</td>
<td>1275</td>
</tr>
<tr>
<td>path</td>
<td>1456</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>649</td>
</tr>
<tr>
<td>sub_match</td>
<td>1503</td>
</tr>
<tr>
<td>sys_days</td>
<td>1252</td>
</tr>
<tr>
<td>sys_info</td>
<td>1297</td>
</tr>
<tr>
<td>sys_time</td>
<td>1251</td>
</tr>
<tr>
<td>sys_time</td>
<td>1305</td>
</tr>
<tr>
<td>tai_time</td>
<td>1255</td>
</tr>
<tr>
<td>thread::id</td>
<td>1560</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>639</td>
</tr>
<tr>
<td>utc_time</td>
<td>1253</td>
</tr>
<tr>
<td>valarray</td>
<td>1199</td>
</tr>
<tr>
<td>weekday</td>
<td>1270</td>
</tr>
<tr>
<td>weekday_indexed</td>
<td>1271</td>
</tr>
<tr>
<td>weekday_last</td>
<td>1272</td>
</tr>
<tr>
<td>year</td>
<td>1267</td>
</tr>
<tr>
<td>year_month</td>
<td>1277</td>
</tr>
<tr>
<td>year_month_day</td>
<td>1280</td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1283</td>
</tr>
<tr>
<td>year_month_weekday</td>
<td>1285</td>
</tr>
<tr>
<td>year_month_weekday_last</td>
<td>1287</td>
</tr>
<tr>
<td>zoned_time</td>
<td>1303</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>1664</td>
</tr>
<tr>
<td>duration</td>
<td>1245</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1021</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>984</td>
</tr>
<tr>
<td>leap_second</td>
<td>1305</td>
</tr>
<tr>
<td>move_iterator</td>
<td>941</td>
</tr>
<tr>
<td>optional</td>
<td>591, 592</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>521, 522</td>
</tr>
<tr>
<td>queue</td>
<td>889</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>934</td>
</tr>
<tr>
<td>stack</td>
<td>894</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>524</td>
</tr>
<tr>
<td>sys_time</td>
<td>1305</td>
</tr>
<tr>
<td>time_point</td>
<td>1250</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>997</td>
</tr>
<tr>
<td>type_index</td>
<td>716</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>638, 639</td>
</tr>
<tr>
<td>valarray</td>
<td>1200, 1201</td>
</tr>
<tr>
<td>variant</td>
<td>602</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>523</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator&lt;&lt;&gt;</td>
<td></td>
</tr>
<tr>
<td>basic_string_view</td>
<td>779</td>
</tr>
<tr>
<td>coroutine_handle</td>
<td>530</td>
</tr>
<tr>
<td>counted_iterator</td>
<td>951</td>
</tr>
<tr>
<td>day</td>
<td>1263</td>
</tr>
<tr>
<td>directory_entry</td>
<td>1465</td>
</tr>
<tr>
<td>duration</td>
<td>1245</td>
</tr>
<tr>
<td>elements_view::iterator</td>
<td>1022</td>
</tr>
<tr>
<td>error_category</td>
<td>555</td>
</tr>
<tr>
<td>error_code</td>
<td>559</td>
</tr>
<tr>
<td>error_condition</td>
<td>559</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>984</td>
</tr>
<tr>
<td>leap_second</td>
<td>1305</td>
</tr>
<tr>
<td>monostate</td>
<td>603</td>
</tr>
<tr>
<td>month</td>
<td>1265</td>
</tr>
<tr>
<td>month_day</td>
<td>1273</td>
</tr>
<tr>
<td>month_day_last</td>
<td>1274</td>
</tr>
<tr>
<td>move_iterator</td>
<td>942</td>
</tr>
<tr>
<td>optional</td>
<td>591, 592</td>
</tr>
<tr>
<td>pair</td>
<td>570</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>522</td>
</tr>
<tr>
<td>path</td>
<td>1457</td>
</tr>
<tr>
<td>queue</td>
<td>890</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>935</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>648</td>
</tr>
<tr>
<td>stack</td>
<td>894</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>524</td>
</tr>
<tr>
<td>sub_match</td>
<td>1502, 1503</td>
</tr>
<tr>
<td>sys_time</td>
<td>1305</td>
</tr>
<tr>
<td>thread::id</td>
<td>1560</td>
</tr>
<tr>
<td>time_zone</td>
<td>1299</td>
</tr>
<tr>
<td>time_zone_link</td>
<td>1306</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>997</td>
</tr>
<tr>
<td>tuple</td>
<td>580</td>
</tr>
<tr>
<td>type_index</td>
<td>716</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>638, 639</td>
</tr>
<tr>
<td>variant</td>
<td>602</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>523</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>1267</td>
</tr>
<tr>
<td>year_month</td>
<td>1277</td>
</tr>
<tr>
<td>year_month_day</td>
<td>1280</td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1283</td>
</tr>
<tr>
<td>year_month_weekday</td>
<td>1285</td>
</tr>
<tr>
<td>year_month_weekday_last</td>
<td>1287</td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1282</td>
</tr>
<tr>
<td>year_month</td>
<td>1277</td>
</tr>
<tr>
<td>year_month_day</td>
<td>1279</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Line Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>operator=</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>629</td>
</tr>
<tr>
<td>any</td>
<td>606, 607</td>
</tr>
<tr>
<td>atomic</td>
<td>1535</td>
</tr>
<tr>
<td>atomic&lt;floating-point&gt;</td>
<td>1535</td>
</tr>
<tr>
<td>atomic&lt;integral&gt;</td>
<td>1535</td>
</tr>
<tr>
<td>atomic&lt;shared_ptr&lt;T&gt;&gt;</td>
<td>1545</td>
</tr>
<tr>
<td>atomic&lt;T*&gt;</td>
<td>1535</td>
</tr>
<tr>
<td>atomic&lt;weak_ptr&lt;T&gt;&gt;</td>
<td>1547</td>
</tr>
<tr>
<td>atomic_ref</td>
<td>1528</td>
</tr>
<tr>
<td>atomic_ref&lt;floating-point&gt;</td>
<td>1528</td>
</tr>
<tr>
<td>atomic_ref&lt;integral&gt;</td>
<td>1528</td>
</tr>
<tr>
<td>atomic_ref&lt;T*&gt;</td>
<td>1528</td>
</tr>
<tr>
<td>back_insert_iterator</td>
<td>936</td>
</tr>
<tr>
<td>basic_filebuf</td>
<td>1423</td>
</tr>
<tr>
<td>basic_iostream</td>
<td>1394</td>
</tr>
<tr>
<td>basic_istream</td>
<td>1386</td>
</tr>
</tbody>
</table>
Index of library names

1793
transform_view::iterator, 996
transform_view::sentinel, 998
tuple, 580
type_index, 716
type_info, 512
unique_ptr, 638
unreachable_sentinel_t, 951
valarray, 1200, 1201
variant, 602
vector, 842
weak_ordering, 523
weekday, 1269
weekday_indexed, 1271
weekday_last, 1272
year, 1267
year_month, 1276
year_month_day, 1279
year_month_day_last, 1282
year_month_weekday, 1284
year_month_weekday_last, 1287
zoned_time, 1303
operator>, 1664
duration, 1245
elements_view::iterator, 1021
iota_view::iterator, 984
leap_second, 1305
move_iterator, 941
optional, 591, 592
partial_ordering, 521, 522
queue, 889
reverse_iterator, 934
stack, 894
strong_ordering, 524
sys_time, 1305
time_point, 1250
transform_view::iterator, 997
type_index, 716
unique_ptr, 638, 639
valarray, 1200, 1201
variant, 602
weak_ordering, 523
operator>>, 1664
duration, 1245
elements_view::iterator, 1022
iota_view::iterator, 984
leap_second, 1305
move_iterator, 941
optional, 591, 592
partial_ordering, 521, 522
queue, 889
reverse_iterator, 934
stack, 894
strong_ordering, 524
sys_time, 1305
time_point, 1250
transform_view::iterator, 997
type_index, 716
unique_ptr, 638, 639
valarray, 1200, 1201
variant, 602
weak_ordering, 523
operator>>

basic_istream, 1387–1389, 1393
basic_string, 767
bitset, 613, 614
byte, 491
complex, 1147
path, 1457
valarray, 1199, 1200
operator>>=

bitset, 612
byte, 491
gslice_array, 1206
indirect_array, 1208
mask_array, 1207
slice_array, 1203
valarray, 1198
operator[]

basic_string, 756
basic_string_view, 774
bitset, 613, 614
counted_iterator, 949
indirect_array, 1207
iota_view::iterator, 984
map, 852
mask_array, 1206
match_results, 1506
move_iterator, 940
reverse_iterator, 933
shared_ptr, 644
span, 899
unique_ptr, 637
unordered_map, 871
valarray, 1196, 1197
weekday, 1269
operator%
duration, 1244
valarray, 1199, 1200
operator%=
duration, 1243
gslice_array, 1206
indirect_array, 1208
mask_array, 1207
slice_array, 1203
valarray, 1198
operator&
bitset, 614
byte, 491
valarray, 1199, 1200
operator&=
atomic<integral>, 1539
atomic_ref<integral>, 1531
bitset, 611
byte, 491
gslice_array, 1206
indirect_array, 1208
mask_array, 1207
slice_array, 1203
valarray, 1198
operator&&
valarray, 1200, 1201
operator~
  bitset, 614
  byte, 492
  valarray, 1199, 1200
operator=
  atomic<integral>, 1539
  atomic_ref<integral>, 1531
  bitset, 612
  byte, 491
  gslice_array, 1206
  indirect_array, 1208
  mask_array, 1207
  slice_array, 1203
  valarray, 1198
operator~
  bitset, 612
  byte, 492
  valarray, 1197
operator--
  counted_iterator, 950
operator|
  bitset, 614
  byte, 491
  valarray, 1199, 1200
operator|=   
  atomic<integral>, 1539
  atomic_ref<integral>, 1531
  bitset, 611
  byte, 491
  gslice_array, 1206
  indirect_array, 1208
  mask_array, 1207
  slice_array, 1203
  valarray, 1198
operator||
  valarray, 1200, 1201
optimize
  syntax_option_type, 1492, 1493
optional, 582
  constructor, 583–585
  destructor, 585
  emplace, 588
  has_value, 589
  hash, 593
  operator bool, 589
  operator!=, 591, 592
  operator*, 589
  operator->, 589
  operator<, 591, 592
  operator<=, 591, 592
  operator==, 585–587
  operator==, 590, 591
  operator>, 591, 592
  operator>=, 591, 592
  reset, 590
  swap, 588, 592
  value, 589, 590
  value_or, 590
  value_type, 582
options   
  recursive_directory_iterator, 1469
  synchronized_pool_resource, 660
  unsynchronized_pool_resource, 660
ostream, 1356, 1383
ostream_iterator, 953
  constructor, 954
  operator*, 954
  operator++, 954
  operator*, 954
ostreambuf_iterator, 956, 1356
  constructor, 957
  failed, 957
  operator*, 957
  operator++, 957
  operator*, 957
ostringstream, 1356, 1407
ostream, 1670
  constructor, 1670
  freeze, 1671
  pcount, 1671
  rdbuf, 1671
  str, 1671
osyncstream, 1356, 1433
out
  basic_format_context, 734
  codecvt, 1329
  format_to_n_result, 721
out_of_range, 547, 548, 611–613
  constructor, 549
outer_allocator
  scoped_allocator_adaptor, 665
outer_allocator_type
  scoped_allocator_adaptor, 662
output_iterator, 919
output_iterator_tag, 928
output_range, 971
overflow
  basic_filebuf, 1425
  basic_streambuf, 1382
  basic_stringbuf, 1412
  strstreambuf, 1667
overflow_error, 547, 549, 611, 613
  constructor, 549, 550
owner_before
  shared_ptr, 644
  weak_ptr, 651
owner_less, 651
  operator(), 652
owns_lock
  shared_lock, 1581
  unique_lock, 1578
Index of library names

bernoulli_distribution, 1175
binomial_distribution, 1176
geometric_distribution, 1177
negative_binomial_distribution, 1177
packaged_task, 1607
bernoulli_distribution, 1175
binomial_distribution, 1176
destructor, 1608
get_future, 1608
make_ready_at_thread_exit, 1608
operator(), 1608
operator=, 1608
reset, 1609
swap, 1608, 1609
valid, 1608
constructor, 1448
copy, 1471
directory_entry, 1464
destructor, 1608
directory_entry, 1464
empty, 1454
end, 1456
extension, 1454
extension, 1454
filename, 1453
filename, 1453
generic_string, 1452, 1453
generic_u16string, 1453
generic_u32string, 1453
generic_u8string, 1453
has_extension, 1455
has_extension, 1455
has_filename, 1455
has_filename, 1455
has_parent_path, 1455
has_relative_path, 1455
has_root_directory, 1454
has_root_directory, 1454
has_root_path, 1454
has_root_path, 1454
has_stem, 1455
hash_value, 1457
is_absolute, 1455
is_absolute, 1455
iterator, 1456
iterator, 1456
lexically_normal, 1455
lexically_normal, 1455
lexically_proximate, 1456
lexically_proximate, 1456
lexically_relative, 1455
lexically_relative, 1455
make_preferred, 1451
make_preferred, 1451
operator string_type, 1452
operator string_type, 1452
operator+=, 1450, 1451
operator+=, 1450, 1451
operator/, 1457
operator/, 1457
operator<, 1456
operator<, 1456
operator<=, 1457
operator<=, 1457
operator==, 1449
operator==, 1449
operator==, 1457
operator==, 1457
parent_path, 1453
parent_path, 1453
preferred_separator, 1445
preferred_separator, 1445
relative_path, 1453
relative_path, 1453
remove, 1479
remove_filename, 1451
replace_extension, 1452
replace_extension, 1452
replace_filename, 1451
replace_filename, 1451
root_directory, 1453
root_directory, 1453
root_name, 1453
root_name, 1453
root_path, 1453
root_path, 1453
stem, 1454
stem, 1454
string, 1452
string, 1452
swap, 1452, 1457
swap, 1452, 1457
u16string, 1452
u16string, 1452
u32string, 1452
u32string, 1452
u8string, 1452
u8string, 1452
value_type, 1445
value_type, 1445
wstring, 1452
wstring, 1452
append, 1450
assign, 1449
begin, 1456
begin, 1456
c_str, 1452
c_str, 1452
clear, 1451
clear, 1451
compare, 1453
compare, 1453
concat, 1450, 1451
concat, 1450, 1451
filesystem_error, 1458
filesystem_error, 1458
path1
path2
filesystem_error, 1458
pbackfail
  basic_filebuf, 1425
  basic_streambuf, 1382
  basic_stringbuf, 1411
  strstreambuf, 1668
pbase
  basic_streambuf, 1380
pbump
  basic_streambuf, 1380
pcount
  ostrstream, 1671
  strstream, 1672
  strstreambuf, 1667
peek
  basic_istream, 1392
perm_options, 1460
permissions, 1478
  file_status, 1461, 1462
perms, 1460
permutable, 927
perror, 1482
piecewise_constant_distribution, 1187
  constructor, 1187, 1188
  densities, 1188
  intervals, 1188
  result_type, 1187
piecewise_construct, 571
piecewise_construct_t, 571
piecewise_linear_distribution, 1188
  constructor, 1189
  densities, 1190
  intervals, 1189
  result_type, 1188
placeholders, 681
plus, 671
  operator(), 671
  plus<>, 671
    operator(), 671
pmr::string
  hash, 770
pmr::u16string
  hash, 770
pmr::u32string
  hash, 770
pmr::wstring
  hash, 770
pointer
  allocator_traits, 628
  basic_string, 747
  basic_string_view, 771
  iterator_traits, 911
  scoped_allocator_adaptor, 662
pointer_safety
  preferred, 615
  relaxed, 615
  strict, 615
pointer_to
  pointer_traits, 623
pointer_to_binary_function
  zombie, 481
pointer_to_unary_function
  zombie, 481
pointer_traits, 622
  difference_type, 622
  element_type, 622
  pointer_to, 623
  rebind, 622
  to_address, 623
poisson_distribution, 1177
  constructor, 1178
  mean, 1178
  result_type, 1177
polar
  complex, 1148
polymorphic_allocator, 655
  allocate, 656
  allocate_bytes, 656
  allocate_object, 656
  construct, 657
  constructor, 656
  deallocate, 656
  deallocate_bytes, 656
  deallocate_object, 656
  destroy, 657
  new_object, 656, 657
  operator==, 657
  resource, 657
  select_on_container_copy_construction, 657
  value_type, 655
pool_options, 658
  largest_required_pool_block, 659
  max_blocks_per_chunk, 659
pop
  forward_list, 831
  priority_queue, 892
  recursive_directory_iterator, 1470
pop_back
  basic_string, 760
pop_heap, 1112
popcount, 1153
position
  match_results, 1506
  positive_sign
    moneypunct, 1351
pow, 1150, 1209
  complex, 1149
  valarray, 1201
powf, 1209
powl, 1209
pptr
  basic_streambuf, 1380
precision
  ios_base, 1322, 1365
pred
  drop_while_view, 1004
  filter_view, 990
take_while_view, 1001
predicate, 545
preferred
pointer_safety, 615
preferred_separator
path, 1445
prefix
match_results, 1506
prev, 929, 931
subrange, 976
prev_permutation, 1121
PRIuFASTN, 1483
PRIuLEASTN, 1483
PRIuMAX, 1483
PRIuN, 1483
PRIuPTR, 1483
PRIuFASTN, 1483
PRIuLEASTN, 1483
PRIuMAX, 1483
PRIuN, 1483
PRIuPTR, 1483
printf, 1482
PRIoFASTN, 1483
PRIoLEASTN, 1483
PRIoMAX, 1483
PRIoN, 1483
PRIoPTR, 1483
priority_queue, 890
constructor, 891, 892
emplace, 892
pop, 892
swap, 892
PRIuFASTN, 1483
PRIuLEASTN, 1483
PRIuMAX, 1483
PRIuN, 1483
PRIuPTR, 1483
PRIxFASTN, 1483
PRIxLEASTN, 1483
PRIxMAX, 1483
PRIxN, 1483
PRIxPTR, 1483
PRIxFASTN, 1483
PRIxLEASTN, 1483
PRIxMAX, 1483
PRIxN, 1483
PRIxPTR, 1483
probabilities
discrete_distribution, 1186
proj
complex, 1148
projected, 926
promise, 1598
constructor, 1599
coroutine_handle, 530
coroutine_handle<noop_coroutine_
        -promise>, 531
destructor, 1599
get_future, 1599
operator=, 1599
set_exception, 1600
set_exception_at_thread_exit, 1600
set_value, 1600
set_value_at_thread_exit, 1600
swap, 1599, 1600
uses_allocator, 1599
propagate_on_container_copy_assignment
allocator_traits, 628
scoped_allocator_adaptor, 664
propagate_on_container_move_assignment
allocator, 629
allocator_traits, 628
scoped_allocator_adaptor, 664
propagate_on_container_swap
allocator_traits, 628
scoped_allocator_adaptor, 664
proximate, 1478, 1479
proxy
istreambuf_iterator, 955
ptr
from_chars_result, 719
to_chars_result, 719
ptr_fun
zombie, 481
ptrdiff_t, 488
pubimbue
basic_streambuf, 1378
pubseekoff
basic_streambuf, 1378
pubseekpos
basic_streambuf, 1378
pubsetbuf
basic_streambuf, 1378
pubsync
basic_streambuf, 1378
push
priority_queue, 892
push_back
basic_string, 757
deque, 827
push_front
deque, 827
forward_list, 831
push_heap, 1112
put
basic_ostream, 1401
money_put, 1349
num_put, 1336
time_put, 1346
put_money, 1404
put_time, 1405
putback
basic_istream, 1392
putc, 1482
putchar, 1482
putenv, 532

Index of library names
Index of library names
Index of library names

unordered associative containers, 817
reinterpret_pointer_cast
shared_ptr, 649
rel_ops, 1664
rel_ops, 1664
relation, 545
relative, 1479
relative_path
path, 1453
relaxed
memory_order, 1524
pointer_safety, 615
release
counting_semaphore, 1591
memory_order, 1524
monotonic_buffer_resource, 662
shared_lock, 1581
synchronized_pool_resource, 660
unique_lock, 1578
unique_ptr, 635
unsynchronized_pool_resource, 660
reload_tzdb, 1295
remainder, 1209
remainderf, 1209
remainderl, 1209
remote_version, 1295
remove, 1087, 1482
forward_list, 833
list, 838
path, 1479
remove_all, 1479
remove_all_extents, 707
remove_all_extents_t, 692
remove_const, 706
remove_const_t, 691
remove_copy, 1088
remove_copy_if, 1088
remove_cv, 706
remove_cv_t, 691
remove_cvref, 708
remove_cvref_t, 692
remove_extent, 707
remove_extent_t, 692
remove_filename
path, 1451
remove_if, 1087
forward_list, 833
remove_pointer, 708
remove_pointer_t, 692
remove_prefix
basic_string_view, 775
remove_reference, 706
remove_reference_t, 691
remove_suffix
basic_string_view, 775
remove_volatile, 706
remove_volatile_t, 691
remquo, 1209
remquof, 1209
remquol, 1209

Index of library names

reference
basic_string, 747
basic_string_view, 771
iterator_traits, 911
reference_wrapper, 669
constructor, 670
cref, 671
get, 670
operator T&, 670
operator(), 670
operator=, 670
ref, 671
refresh
directory_entry, 1464
regex, 1488
regex_constants, 1492
error_type, 1494, 1495
match_flag_type, 1492
syntax_option_type, 1492
regex_error
constructor, 1495
regex_iterator
constructor, 1512
increment, 1513
operator*, 1513
operator++, 1513, 1514
operator->, 1513
operator==, 1513
regex_match, 1508, 1509
regex_replace, 1511
regex_search, 1509, 1510
regex_token_iterator, 1514
constructor, 1516
end-of-sequence, 1514
operator*, 1516
operator++, 1517
operator->, 1516
operator==, 1514, 1516
regex_traits, 1495
char_class_type, 1496
isctype, 1496
length, 1496
lookup_classname, 1496
lookup_collatename, 1496
transform, 1496
transform_primary, 1496
translate, 1496
translate_nocase, 1496
value, 1497
register_callback
io_base, 1367
regular, 545
regular_expression_traits
isctype, 1518
lookup_classname, 1518
lookup_collatename, 1518
transform_primary, 1518
regular_invocable, 545
rehash
rename, 1480, 1482
rend
  basic_string, 755
  basic_string_view, 774
span, 900
rend(C&), 958
rend(initializer_list<E>), 958
rend(T (&array)[N]), 958
rep
  system_clock, 1251
replace, 1084
  basic_string, 760–762
replace_copy, 1085
replace_copy_if, 1085
replace_extension
  path, 1452
replace_filename
  directory_entry, 1463
  path, 1451
replace_if, 1084
request_stop
  jthread, 1564
  stop_source, 1557
required_alignment
  atomic_ref, 1528
  atomic_ref<floating-point>, 1528
  atomic_ref<integral>, 1528
  atomic_ref<T>, 1528
reserve
  basic_string, 755, 1676
  unordered associative containers, 817
  vector, 842
reset
  any, 607
  bitset, 612
  optional, 590
  packaged_task, 1609
  shared_ptr, 643, 644
  unique_ptr, 635, 637
  weak_ptr, 651
resetiosflags, 1402
resize
  basic_string, 755
  deque, 826
  forward_list, 832
  list, 837
  valarray, 1199
  vector, 843
resize_file, 1480
resource
  polymorphic_allocator, 657
result
  local_info, 1298
result_of
  zombie, 481
result_of_t
  zombie, 481
result_type
  bernoulli_distribution, 1175
  binomial_distribution, 1175
  cauchy_distribution, 1183
  chi_squared_distribution, 1183
  discard_block_engine, 1167
  discrete_distribution, 1185
  exponential_distribution, 1178
  extreme_value_distribution, 1180
  fisher_distribution, 1184
  function, 682
  gamma_distribution, 1179
  geometric_distribution, 1176
  independent_bits_engine, 1168
  linear_congruential_engine, 1163
  lognormal_distribution, 1182
  mersenne_twister_engine, 1164
  negative_binomial_distribution, 1177
  normal_distribution, 1181
  piecewise_constant_distribution, 1187
  piecewise_linear_distribution, 1188
  poisson_distribution, 1177
  random_device, 1170
  seed_seq, 1171
  shuffle_order_engine, 1168
  student_t_distribution, 1185
  subtract_with_carry_engine, 1165
  uniform_int_distribution, 1173
  uniform_real_distribution, 1174
  weibull_distribution, 1180
resume
  coroutine_handle, 530
  coroutine_handle<noop_coroutine_promise>, 531
rethrow_exception, 517
rethrow_if_nested
  nested_exception, 518
rethrow_nested
  nested_exception, 518
return_temporary_buffer
  zombie, 481
reverse, 1091
  forward_list, 834
  list, 839
reverse_copy, 1091
reverse_iterator, 931
  base, 933
  basic_string, 747
  basic_string_view, 771
  constructor, 932, 933
  iter_move, 935
  iter_swap, 935
make_reverse_iterator non-member
  function, 935
operator!=, 934
operator*, 933
operator*, 933, 935
operator**, 933
operator**, 934
operator-, 933, 935
Index of library names

operator-=', 934
operator->, 933
operator--=, 933, 934
operator<, 934
operator<<, 934
operator|, 935
operator=, 933
operator==, 934
operator>, 934
operator|>, 934
operator[], 933
reverse_view, 1017
base, 1017
begin, 1017
constructor, 1017
end, 1017
size, 1017
 rewind, 1482
rfind
 basic_string, 763
 basic_string_view, 777
riemann_zeta, 1222
riemann_zetaf, 1222
riemann_zetal, 1222
right, 1374
rint, 1209
rintf, 1209
rintl, 1209
root_directory
 path, 1453
root_name
 path, 1453
root_path
 path, 1453
rotate, 1092
rotate_copy, 1092
rotr, 1152
rotl, 1152
round, 1209
duration, 1246
time_point, 1250
round_error
 numeric_limits, 497
round_indeterminate, 495
round_style
 numeric_limits, 499
round_to_nearest, 495
round_toward_infinity, 495
round_toward_neg_infinity, 495
round_toward_zero, 495
roundf, 1209
roundl, 1209
runtime_error, 547, 549
 constructor, 549
S
 s
 lognormal_distribution, 1182
same_as, 537
sample, 1093
save
 sys_info, 1297
 sbumpc
 basic_streambuf, 1378
scalbln, 1209
scalblnf, 1209
scalblnl, 1209
scalbn, 1209
scalbnf, 1209
scalbnl, 1209
scan_is
 ctype, 1324
ctype<char>, 1327
scan_not
 ctype, 1324
ctype<char>, 1327
scanf, 1482
SCHAR_MAX, 501
SCHAR_MIN, 501
scientific, 1374
 chars_format, 718
SCNdFASTN, 1483
SCNdLEASTN, 1483
SCNdMAX, 1483
SCNdN, 1483
SCNdPTR, 1483
SCNdFASTN, 1483
SCNdLEASTN, 1483
SCNdMAX, 1483
SCNdN, 1483
SCNdPTR, 1483
SCNuFASTN, 1483
SCNuLEASTN, 1483
SCNuMAX, 1483
SCNuN, 1483
SCNuPTR, 1483
SCNuFASTN, 1483
SCNuLEASTN, 1483
SCNuMAX, 1483
SCNuN, 1483
SCNuPTR, 1483
SCNxFASTN, 1483
SCNxLEASTN, 1483
SCNxMAX, 1483
SCNxN, 1483
SCNxPTR, 1483
scoped_allocator_adaptor, 662
 allocate, 665
 const_pointer, 662
 const_void_pointer, 662
 construct, 665
 constructor, 664
deallocate, 665
destroy, 665
difference_type, 662
inner_allocator, 665
inner_allocator_type, 664
is_always_equal, 664
max_size, 665
operator==, 665
outer_allocator, 665
outer_allocator_type, 662
pointer, 662
propagate_on_container_copy_assignment, 664
propagate_on_container_move_assignment, 664
propagate_on_container_swap, 664
select_on_container_copy_construction, 665
size_type, 662
value_type, 662
void_pointer, 662
scoped_lock, 1574
constructor, 1574
destructor, 1575
search, 1077, 1078
search_n, 1077, 1078
second
  local_info, 1298
second_argument_type
  zombie, 482
seconds, 1225
  hh_mm_ss, 1292
seed_seq, 1171
  constructor, 1171, 1172
generate, 1172
dparam, 1172
  result_type, 1171
set
  1172
 SEEK_CUR, 1482
seek_dir
  zombie, 482
SEEK_END, 1482
SEEK_SET, 1482
seekdir
  ios_base, 1364
seekg
  basic_istream, 1393
seekoff
  basic_filebuf, 1426
  basic_streambuf, 1380
  basic_stringbuf, 1412
  strstreambuf, 1668
seekp
  basic_ostream, 1398
seekpos
  basic_filebuf, 1426
  basic_streambuf, 1380
  basic_stringbuf, 1413
  strstreambuf, 1669
select_on_container_copy_construction
  allocator_traits, 629
  polymorphic_allocator, 657
  scoped_allocator_adaptor, 665
  semiregular, 545
sentinel
  filter_view, 992
  sentinel_for, 918
sentry
  basic_istream, 1386
  basic_ostream, 1397
  constructor, 1386
destructor, 1387
seq, 718
  execution, 718
seq_cst
  memory_order, 1524
set, 858
  clear, 805
  constructor, 799, 860
contains, 805
count, 805
emplace, 801
emplace_hint, 801
equal_range, 806
erase, 805
erase_if, 861
extract, 804
find, 805
insert, 801
key_comp, 800
key_compare, 799
key_type, 799
lower_bound, 806
mapped_type, 799
merge, 804
node_type, 799
operator<, 860
operator==, 860
upper_bound, 806
value_comp, 800
value_compare, 799
value_type, 799
set (member)
basic_ostream, 1397
clear, 805
destructor, 799, 860
contains, 805
count, 805
destructor, 860
set_difference, 1110
set emission on sync
  basic_osyncstream, 1436, 1437
  basic_syncstream, 1435
set_exception
  promise, 1600
set_exception_at_thread_exit
  promise, 1600
set_intersection, 1109
set_new_handler, 483, 510
set_rdbuf
  basic_ios, 1372
set_symmetric_difference, 1111
set_terminate, 483, 516
set_unexpected
  zombie, 481
Index of library names

SIG_IGN, 532
SIGABRT, 532
SIGFPE, 532
SIGILL, 532
SIGINT, 532
signal, 532
signaling_NaN
    numeric_limits, 499
signbit
    complex, 1149
    valarray, 1201
sin, 1209
    complex, 1149
    valarray, 1201
sinf, 1209
single_view, 978
    begin, 979
    constructor, 979
    data, 979
    end, 979
    size, 979
sinh, 1209
    complex, 1149
    valarray, 1201
sinhf, 1209
sinhl, 1209
sinl, 1209
size
    array, 821, 823
    basic_string, 755
    basic_string_view, 774
    bitset, 613
    common_view, 1015
    drop_view, 1002
    elements_view, 1018
    format_to_n_result, 721
    gslice, 1205
    initializer_list, 519
    iota_view, 981
    match_results, 1505
    reverse_view, 1017
    seed_seq, 1172
    single_view, 979
    slice, 1202
    span, 899
    subrange, 976
    take_view, 999
    transform_view, 993
    valarray, 1198
size(C& c), 958
size(T (&array)[N]), 958
size_bytes
    span, 899
size_t, 123, 488, 489, 780–782, 1313, 1482
size_type
    allocator, 629
    allocator_traits, 628
    basic_string, 747
    basic_string_view, 771
    scoped_allocator_adaptor, 662
sized_range, 970
sized_sentinel_for, 918
skipws, 1373
sleep_for
    this_thread, 1565
sleep_until
    this_thread, 1565
slice, 1202
    constructor, 1202
    size, 1202
    start, 1202
    stride, 1202
slice_array, 1203
    operator*=, 1203
    operator+=, 1203
    operator-=, 1203
    operator/=, 1203
    operator%<, 1203
    operator%=, 1203
    operator%>, 1203
    operator%>=, 1203
    value_type, 1203
snextc
    basic_streambuf, 1378
snprintf, 1482
sort, 1095
    forward_list, 834
    list, 839
sortable, 928
source_location, 513
space, 1480
span, 895
    back, 899
    begin, 899
    constructor, 896, 897
    data, 899
    deduction guide, 898
    empty, 899
    end, 899
    first, 898
    front, 899
    iterator, 899
    last, 898, 899
    operator=, 898
    rbegin, 900
    rend, 900
    size, 899
    size_bytes, 899
    subsrange, 898, 899
    subslice, 1223
sph_bessel, 1223
sph_besself, 1223
sph_bessell, 1223
sph_legendre, 1223
slice, 1202
starts_with
  basic_string, 764
  basic_string_view, 776
state
  fpos, 1368
  wbuffer_convert, 1680
  wstring_convert, 1679
state_type
  char_traits, 741
  wbuffer_convert, 1681
  wstring_convert, 1679
static_pointer_cast
  shared_ptr, 648
status, 1480
  directory_entry, 1465
status_known, 1481
stddev
  normal_distribution, 1182
stderr, 1482
stdin, 1482
stdout, 1482
steady_clock, 1258
stem
  path, 1454
  stod, 768, 769
  stof, 768, 769
  stoi, 768, 769
  stol, 768, 769
  stold, 768, 769
  stoll, 768, 769
stop_callback, 1557
  constructor, 1558
  destructor, 1558
stop_possible
stop_token
stopRequested
stop_token
stop_possible
stop_token
stop_possible
stop_token
stop_possible
stop_token
stop_possible
stop_token
stop_possible
stop_token
stop_possible
stop_token
stop_possible
stop_token
store
atomic, 1535
atomic<floating-point>, 1535
atomic<integral>, 1535
atomic<shared_ptr<T>>, 1544
atomic<T*>, 1535
atomic_ref, 1528
atomic<floating-point>, 1535
atomic<floating-point>, 1528
atomic<integral>, 1528
atomic_ref<T*>, 1528
atomic_ref<T>, 1528

stossc
zombie, 482
stoul, 768, 769
stoull, 768, 769

str
basic_istringstream, 1415
basic_ostringstream, 1418
basic_stringbuf, 1411
basic_stringstream, 1420, 1421
istrstream, 1670
match_results, 1506
ostrstream, 1671
strstream, 1672
strstreambuf, 1667
sub_match, 1502
strcat, 780
strchr, 780
strcspn, 780
strcoll, 780
strcpy, 780
strcspn, 780
streambuf, 1356, 1375
streamoff, 1360, 1368
streampos, 1356
streamsize, 1361
strerror, 780
strerror, 1313, 1346
strict
pointer_safety, 615
strict_weak_order, 546
stride
slice, 1205
slice, 1202
string
hash, 770
operator"s, 770
path, 1452
string_view
hash, 779
operator"sv, 779
stringbuf, 1356, 1407
stringstream, 1356, 1407
strlen, 780, 1667, 1670
strncat, 780
strncpy, 780
strncpy, 780
strong_order, 526
strong_ordering, 523
equal, 523
equivalent, 523
greater, 523
less, 523
operator partial_ordering, 524
operator weak_ordering, 524
operator<, 524
operator<=, 524
operator<>, 524
operator<=, 524
operator>, 524
operator>=, 524
strpbrk, 780
strrchr, 780
strspn, 780
strstr, 780
strstream, 1671
constructor, 1671
destructor, 1671, 1672
freeze, 1672
pcount, 1672
rbuff, 1672
str, 1672
strstreambuf, 1665
constructor, 1666, 1667
destructor, 1667
freeze, 1667
overflow, 1667
pbackfail, 1668
pcount, 1667
seekoff, 1668
seekpos, 1669
setbuf, 1669
str, 1667
underflow, 1668
strtod, 489
strtof, 489
strtoimax, 1483
strtok, 780
strtol, 489
strtof, 489
strtol, 489
strtol, 489
strtol, 489
strtol, 489
strtol, 1483
strxfrm, 780
student_t_distribution, 1185
constructor, 1185
mean, 1185
result_type, 1185
sub_match, 1501
compare, 1502
constructor, 1502
length, 1502
operator basic_string, 1502
operator<, 1503
operator<, 1502, 1503
operator==, 1502, 1503
str, 1502

Index of library names
Index of library names

subrange, 973
advance, 977
begin, 976
constructor, 975, 976
empty, 976
end, 976
get, 977
next, 976
operator PairLike, 976
prev, 976
size, 976
subseconds
hh_mm_ss, 1292
subspan
span, 898, 899
substr
basic_string, 763
basic_string_view, 775
subtract_with_carry_engine, 1165
constructor, 1166
result_type, 1165
suffix
match_results, 1506
sum
valarray, 1199
sungetc
basic_stringbuf, 1379
suspend_always, 531
await_ready, 531
await_resume, 531
suspend_never, 531
await_ready, 531
await_resume, 531
await_suspend, 531
swap, 540, 563, 564, 580
any, 607, 608
array, 823
basic_filebuf, 1423
basic_fstream, 1432
basic_ifstream, 1428
basic_ios, 1372
basic_iostream, 1394
basic_istream, 1386
basic_istreamstream, 1415
basic_ofstream, 1430
basic_ostream, 1397
basic_ostringstream, 1418
basic_regex, 1501
basic_streambuf, 1379
basic_string, 762, 767
basic_string_view, 775
basic_stringbuf, 1410
basic_stringstream, 1420
basic_synclbuf, 1434, 1435
function, 685
jthread, 1564
match_results, 1507
optional, 588, 592
packaged_task, 1608, 1609
pair, 569, 570
path, 1452, 1457
priority_queue, 892
promise, 1599, 1600
queue, 890
shared_lock, 1581
shared_ptr, 643, 648
stack, 894
stop_source, 1556, 1557
stop_token, 1555
thread, 1561, 1562
tuple, 577
unique_lock, 1578
unique_ptr, 635
valarray, 1198, 1202
variant, 600, 603
vector, 843
vector<bool>, 846
weak_ptr, 651
swap(unique_ptr&, unique_ptr&), 638
swap_ranges, 1082
swappable, 540
swappable_with, 540
swprintf, 781
swscanf, 781
 symlink_status, 1481
directory_entry, 1465
sync
basic_filebuf, 1426
basic_istream, 1393
basic_streambuf, 1380
basic_synclbuf, 1435
sync_with_stdio
ios_base, 1366
synchronized_pool_resource, 658
constructor, 659
destructor, 660
do_allocate, 660
do_deallocate, 660
do_is_equal, 660
options, 660
release, 660
upstream_resource, 660
syntax_option_type, 1492
awk, 1492, 1493
basic, 1492, 1493
collate, 1492, 1493, 1518
ECMAScript, 1492, 1493
egrep, 1492, 1493
extended, 1492, 1493
grep, 1492, 1493
icase, 1492, 1493
multiline, 1493
nosubs, 1492, 1493
optimize, 1492, 1493
regex_constants, 1492
sys_days, 1225
three_way_comparable, 525
three_way_comparable_with, 525
throw_with_nested
    nested_exception, 518
tie, 577
    basic_ios, 1370
tuple, 577
time, 1313
time_get, 1342
    date_order, 1343
do_date_order, 1344
do_get, 1345
do_get_date, 1344
do_get_monthname, 1344
do_get_time, 1344
do_get_weekday, 1344
do_get_year, 1345
get, 1343
get_date, 1343
get_monthname, 1343
get_time, 1343
get_weekday, 1343
get_year, 1343
time_get_byname, 1345
time_point, 1248
    ceil, 1250
cConstructor, 1248
floor, 1250
max, 1249
min, 1249
operator+, 1249
operator++, 1249
operator=, 1249
operator-, 1249
operator--, 1249
operator=, 1249
operator<, 1250
operator<=, 1249
operator<=, 1249
operator>, 1250
operator>=, 1249
operator>, 1250
operator>=, 1250
round, 1250
time_point_cast, 1250
time_since_epoch, 1248
time_point_cast, 1250
time_point, 1250
time_put, 1345
do_put, 1346
put, 1346
time_put_byname, 1346
time_since_epoch
time_point, 1248
time_t, 1313
TIME_UTC, 1313
time_zone, 1298
get_info, 1299
name, 1299
operator<=>, 1299
operator==, 1299
to_local, 1290
to_sys, 1290
time_zone_link, 1305
    name, 1306
operator<=>, 1306
operator==, 1306
target, 1306
timed_mutex, 1569	
timespec, 1313	
timespec_get, 1313	
tinyness_before
    numeric_limits, 499
tm, 781, 1313
TMP_MAX, 1482
tmpfile, 1482
tmpnam, 1482
to_address, 623
    pointer_traits, 623
to_array, 823
to_bytes
    wstring_convert, 1679
to_chars, 720
to_chars_result, 719
eC, 719
ptr, 719
to_duration
    hh_mm_ss, 1292
to_integer
    byte, 492
to_local
    time_zone, 1299
to_string, 769
    bitset, 613
to_sys
    time_zone, 1299
    utc_clock, 1253
to_time_t
    system_clock, 1251
to ullong
    bitset, 613
to ulong
    bitset, 613
to_utc
    gps_clock, 1256
tai_clock, 1255
to wstring, 769
tolower, 779, 1322
type, 1324
type<char>, 1327
totally_ordered, 544
totally_ordered_with, 544
toupper, 779, 1322
type, 1324
type<char>, 1327
towctrans, 780
towlower, 780
towupper, 780	traits_type
    basic_string, 747
basic_string_view, 771
transform, 1082
collate, 1341
regex_traits, 1496
transform_exclusive_scan, 1130
transform_inclusive_scan, 1131
transform_primary
regex_traits, 1496
transform_reduce, 1126, 1127
transform_view, 993
    base, 993
    begin, 993
    constructor, 993
    end, 993, 994
    size, 993
transform_view::iterator, 994
    base, 996
    constructor, 996
    iter_swap, 997
    iterator, 996
    operator+, 997
    operator++, 996
    operator+=, 996
    operator-, 997
    operator=, 996
    operator<, 996
    operator<=, 997
    operator==, 996
    operator>, 997
    operator>=, 997
transform_view::sentinel, 997
    base, 998
    constructor, 998
    operator<, 998
    operator==, 998
    translate
    regex_traits, 1496
translate_nocase
regex_traits, 1496
traps
    numeric_limits, 499
    treat_as_floating_point, 1239
    treat_as_floating_point_v, 1225
    true_type, 696
truename
    numpunct, 1339
trunc, 1209
truncf, 1209
truncl, 1209
try_acquire
    counting_semaphore, 1592
    try_acquire_for
    counting_semaphore, 1592
    try_acquire_until
    counting_semaphore, 1592
try_emplace
    map, 853
unordered_map, 871, 872
try_lock, 1582
    shared_lock, 1580
    unique_lock, 1577
try_lock_for
    shared_lock, 1581
    unique_lock, 1577
try_lock_until
    shared_lock, 1580
    unique_lock, 1577
try_to_lock, 1573
try_to_lock_t, 1573
try_wait
    latch, 1593	tuple, 571, 572, 824
        constructor, 574, 575
        forward_as_tuple, 577
        get, 579
        make_tuple, 577
        operator<=>, 580
        operator=, 576, 577
        operator==, 580
        swap, 577
        tie, 577
tuple_cat, 578
tuple_element, 570, 579, 824
tuple_element_t, 572
tuple_size, 570, 579, 824
        in general, 578
tuple_size_v, 572
type
    any, 608
        file_status, 1461
type_identity, 708
type_identity_t, 692
type_index, 716
        constructor, 716
        hash, 717
        hash_code, 716
        name, 716
        operator<, 716
        operator<=, 716
        operator<=>, 716
        operator==, 716
        operator>, 716
        operator>=, 716
    type_info, 115, 511, 512
        before, 512
        hash_code, 512
        name, 512
        operator==, 512
tzdb, 1293
    current_zone, 1294
    locate_zone, 1293
tzdb_list, 1294
    begin, 1294
    cbegin, 1294
cend, 1295
    end, 1294

Index of library names
erase_after, 1294
front, 1294
U
u16streampos, 1356
u16string
hash, 770
operator""s, 770
path, 1452
u16string_view
hash, 779
operator""sv, 779
u32streampos, 1356
u32string
hash, 770
operator""s, 770
path, 1452
u32string_view
hash, 779
operator""sv, 779
u8path, 1681
u8string
operator""s, 770
path, 1452
u8string_view
hash, 779
operator""sv, 779
UCHAR_MAX, 501
uflow
basic_filebuf, 1425
basic_streambuf, 1381
uint16_t, 502
uint32_t, 502
uint64_t, 502
uint8_t, 502
uint_fast16_t, 502
uint_fast32_t, 502
uint_fast64_t, 502
uint_fast8_t, 502
uint_least16_t, 502
uint_least32_t, 502
uint_least64_t, 502
uint_least8_t, 502
UINT_MAX, 501
uintmax_t, 502
uintptr_t, 502
ULLONG_MAX, 501
ULONG_MAX, 501
unary_function
zombie, 482
unary_negate
zombie, 482
uncaught_exception
zombie, 482
uncaught_exceptions, 441, 517
undeclare_no_pointers, 624
undeclare_reachable, 623
underflow
basic_filebuf, 1425
basic_streambuf, 1381
basic_stringbuf, 1411
strstreambuf, 1668
underflow_error, 547, 550
constructor, 550
underlying_type, 709
underlying_type_t, 692
unexpected
zombie, 482
unexpected_handler
zombie, 482
unget
basic_istream, 1392
ungetc, 1482
ungetwc, 781
uniform_int_distribution, 1173
a, 1174
b, 1174
constructor, 1174
result_type, 1173
uniform_random_bit_generator, 1157
uniform_real_distribution, 1174
a, 1174
b, 1175
constructor, 1174
result_type, 1174
uninitialized_construct_using_allocator, 627
uninitialized_copy, 1136
uninitialized_copy_n, 1137
uninitialized_default_construct, 1135
uninitialized_default_construct_n, 1135
uninitialized_fill, 1138
uninitialized_fill_n, 1139
uninitialized_move, 1137
uninitialized_move_n, 1138
uninitialized_value_construct, 1136
uninitialized_value_construct_n, 1136
unique, 1089
forward_list, 833
list, 839
local_info, 1298
unique_copy, 1090
unique_lock, 1575
constructor, 1576
destructor, 1576
lock, 1577
mutex, 1578
operator bool, 1578
operator=, 1576
owns_lock, 1578
release, 1578
swap, 1578
try_lock, 1577
try_lock_for, 1577
try_lock_until, 1577
unlock, 1578
unique_ptr, 631, 635, 643
constructor, 632, 633, 636
destructor, 634
get, 635
get_deleter, 635
hash, 653
operator bool, 635
operator*, 634
operator->, 635
operator<, 638
operator<<, 639
operator<=, 638, 639
operator<>, 638, 639
operator=, 634, 637
operator==, 638
operator>, 638, 639
operator[]>, 638, 639
operator[], 638
release, 635
reset, 635, 637
swap, 635
unitbuf, 1373
unlock
  shared_lock, 1581
  unique_lock, 1578
unordered
  partial_ordering, 521
unordered_map, 864, 866
  at, 871
  begin, 816
  bucket, 816
  bucket_count, 816
  bucket_size, 816
  cbegin, 817
  cend, 817
  clear, 815
  const_local_iterator, 809
  constructor, 809, 870, 871
  contains, 816
  count, 816
  emplace, 812
  emplace_hint, 812
  end, 817
  equal_range, 816
  erase, 815
  erase_if, 877
  extract, 814
  find, 815
  hash_function, 811
  hasher, 809
  insert, 812, 877
  key_eq, 811
  key_equal, 809
  key_type, 808
  local_iterator, 809
  load_factor, 817
  max_bucket_count, 816
  max_load_factor, 817
  merge, 815
  node_type, 809
  operator[], 871
  rehash, 817
  reserve, 817
  try_emplace, 871, 872
  value_type, 809
unordered_multimap, 864, 873
  begin, 816
  bucket, 816
  bucket_count, 816
  bucket_size, 816
  cbegin, 817
  cend, 817
  clear, 815
  const_local_iterator, 809
  constructor, 809, 876, 877
  contains, 816
  count, 816
  emplace, 812
  emplace_hint, 812
  end, 817
  equal_range, 816
  erase, 815
  erase_if, 877
  extract, 814
  find, 815
  hash_function, 811
  hasher, 809
  insert, 812, 877
  key_eq, 811
  key_equal, 809
  key_type, 808
  local_iterator, 809
  load_factor, 817
  max_bucket_count, 816
  max_load_factor, 817
  merge, 815
  node_type, 809
  rehash, 817
  reserve, 817
unordered_multiset, 865, 882
  begin, 816
  bucket, 816
  bucket_count, 816
  bucket_size, 816
  cbegin, 817
  cend, 817
  clear, 815
  const_local_iterator, 809
  constructor, 809, 885, 886
  contains, 816
  count, 816
  emplace, 812
  emplace_hint, 812
  end, 817
  equal_range, 816

Index of library names 1813

© ISO/IEC

N4861
<table>
<thead>
<tr>
<th>Function/Concept</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase</td>
<td>815</td>
</tr>
<tr>
<td>erase_if</td>
<td>886</td>
</tr>
<tr>
<td>extract</td>
<td>814</td>
</tr>
<tr>
<td>find</td>
<td>815</td>
</tr>
<tr>
<td>hash_function</td>
<td>811</td>
</tr>
<tr>
<td>hasher</td>
<td>809</td>
</tr>
<tr>
<td>insert</td>
<td>812</td>
</tr>
<tr>
<td>key_eq</td>
<td>811</td>
</tr>
<tr>
<td>key_equal</td>
<td>809</td>
</tr>
<tr>
<td>key_type</td>
<td>808</td>
</tr>
<tr>
<td>load_factor</td>
<td>817</td>
</tr>
<tr>
<td>local_iterator</td>
<td>809</td>
</tr>
<tr>
<td>mapped_type</td>
<td>809</td>
</tr>
<tr>
<td>max_bucket_count</td>
<td>816</td>
</tr>
<tr>
<td>max_load_factor</td>
<td>817</td>
</tr>
<tr>
<td>merge</td>
<td>815</td>
</tr>
<tr>
<td>node_type</td>
<td>809</td>
</tr>
<tr>
<td>rehash</td>
<td>817</td>
</tr>
<tr>
<td>reserve</td>
<td>817</td>
</tr>
<tr>
<td>value_type</td>
<td>809</td>
</tr>
<tr>
<td>unordered_set</td>
<td>865, 877, 878</td>
</tr>
<tr>
<td>begin</td>
<td>816</td>
</tr>
<tr>
<td>bucket</td>
<td>816</td>
</tr>
<tr>
<td>bucket_count</td>
<td>816</td>
</tr>
<tr>
<td>bucket_size</td>
<td>816</td>
</tr>
<tr>
<td>cbegin</td>
<td>817</td>
</tr>
<tr>
<td>cend</td>
<td>817</td>
</tr>
<tr>
<td>clear</td>
<td>815</td>
</tr>
<tr>
<td>const_local_iterator</td>
<td>809</td>
</tr>
<tr>
<td>constructor</td>
<td>809, 881</td>
</tr>
<tr>
<td>contains</td>
<td>816</td>
</tr>
<tr>
<td>count</td>
<td>816</td>
</tr>
<tr>
<td>emplace</td>
<td>812</td>
</tr>
<tr>
<td>emplace_hint</td>
<td>812</td>
</tr>
<tr>
<td>end</td>
<td>817</td>
</tr>
<tr>
<td>equal_range</td>
<td>816</td>
</tr>
<tr>
<td>erase</td>
<td>815</td>
</tr>
<tr>
<td>erase_if</td>
<td>882</td>
</tr>
<tr>
<td>extract</td>
<td>814</td>
</tr>
<tr>
<td>find</td>
<td>815</td>
</tr>
<tr>
<td>hash_function</td>
<td>811</td>
</tr>
<tr>
<td>hasher</td>
<td>809</td>
</tr>
<tr>
<td>insert</td>
<td>812</td>
</tr>
<tr>
<td>key_eq</td>
<td>811</td>
</tr>
<tr>
<td>key_equal</td>
<td>809</td>
</tr>
<tr>
<td>key_type</td>
<td>808</td>
</tr>
<tr>
<td>load_factor</td>
<td>817</td>
</tr>
<tr>
<td>merge</td>
<td>815</td>
</tr>
<tr>
<td>node_type</td>
<td>809</td>
</tr>
<tr>
<td>rehash</td>
<td>817</td>
</tr>
<tr>
<td>reserve</td>
<td>817</td>
</tr>
<tr>
<td>value_type</td>
<td>809</td>
</tr>
<tr>
<td>unreachable_sentinel</td>
<td>901</td>
</tr>
<tr>
<td>unreachable_sentinel_t</td>
<td>951</td>
</tr>
<tr>
<td>operator==</td>
<td>951</td>
</tr>
<tr>
<td>unsetf</td>
<td></td>
</tr>
<tr>
<td>ios_base</td>
<td>1365</td>
</tr>
<tr>
<td>unshift</td>
<td></td>
</tr>
<tr>
<td>codecvt</td>
<td>1329</td>
</tr>
<tr>
<td>unsigned_integral</td>
<td>539</td>
</tr>
<tr>
<td>unsynchronized_pool_resource</td>
<td>658</td>
</tr>
<tr>
<td>constructor</td>
<td>659</td>
</tr>
<tr>
<td>destructor</td>
<td>660</td>
</tr>
<tr>
<td>do_allocate</td>
<td>660</td>
</tr>
<tr>
<td>do_deallocate</td>
<td>660</td>
</tr>
<tr>
<td>do_is_equal</td>
<td>660</td>
</tr>
<tr>
<td>options</td>
<td>660</td>
</tr>
<tr>
<td>release</td>
<td>660</td>
</tr>
<tr>
<td>upstream_resource</td>
<td>660</td>
</tr>
<tr>
<td>unwrap_ref_decay_t</td>
<td>666</td>
</tr>
<tr>
<td>unwrap_reference</td>
<td>709</td>
</tr>
<tr>
<td>upper_bound</td>
<td>1101</td>
</tr>
<tr>
<td>ordered associative containers</td>
<td>806</td>
</tr>
<tr>
<td>uppercase</td>
<td>1373</td>
</tr>
<tr>
<td>upstream_resource</td>
<td></td>
</tr>
<tr>
<td>monotonic_buffer_resource</td>
<td>662</td>
</tr>
<tr>
<td>synchronized_pool_resource</td>
<td>660</td>
</tr>
<tr>
<td>unsynchronized_pool_resource</td>
<td>660</td>
</tr>
<tr>
<td>use_count</td>
<td></td>
</tr>
<tr>
<td>shared_ptr</td>
<td>644</td>
</tr>
<tr>
<td>weak_ptr</td>
<td>651</td>
</tr>
<tr>
<td>use_facet</td>
<td></td>
</tr>
<tr>
<td>locale</td>
<td>1322</td>
</tr>
<tr>
<td>uses_allocator</td>
<td>625</td>
</tr>
<tr>
<td>promise</td>
<td>1599</td>
</tr>
<tr>
<td>uses_allocator&lt;tuple&gt;</td>
<td>580</td>
</tr>
<tr>
<td>uses_allocator_construction_args</td>
<td>625, 626</td>
</tr>
<tr>
<td>uses_allocator_v</td>
<td>615</td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>501</td>
</tr>
<tr>
<td>utc_clock</td>
<td>1252</td>
</tr>
<tr>
<td>from_sys</td>
<td>1253</td>
</tr>
<tr>
<td>now</td>
<td>1252</td>
</tr>
<tr>
<td>to_sys</td>
<td>1253</td>
</tr>
<tr>
<td>utc_seconds</td>
<td>1225</td>
</tr>
<tr>
<td>utc_time</td>
<td>1225</td>
</tr>
<tr>
<td>from_stream</td>
<td>1253</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>1253</td>
</tr>
<tr>
<td>V</td>
<td></td>
</tr>
<tr>
<td>va_arg</td>
<td>532</td>
</tr>
<tr>
<td>va_copy</td>
<td>532</td>
</tr>
<tr>
<td>va_end</td>
<td>482, 532</td>
</tr>
<tr>
<td>va_list</td>
<td>482, 532</td>
</tr>
<tr>
<td>va_start</td>
<td>532</td>
</tr>
<tr>
<td>valarray</td>
<td>1193, 1205</td>
</tr>
<tr>
<td>abs</td>
<td>1201</td>
</tr>
<tr>
<td>acos</td>
<td>1201</td>
</tr>
<tr>
<td>apply</td>
<td>1199</td>
</tr>
<tr>
<td>asin</td>
<td>1201</td>
</tr>
<tr>
<td>atan</td>
<td>1201</td>
</tr>
<tr>
<td>atan2</td>
<td>1201</td>
</tr>
<tr>
<td>begin</td>
<td>1208</td>
</tr>
</tbody>
</table>
Index of library names 1816
Index of library names

wchar_t, 780
wctype_t, 780
weak_from_this
  enable_shared_from_this, 653
weak_order, 526
weak_ordering, 522
  equivalent, 522
  greater, 522
  less, 522
  operator partial_ordering, 522
  operator<, 523
  operator<=, 523
  operator==, 523
  operator>, 523
  operator>=, 523
weak_ptr, 642, 649, 653
  constructor, 650
  destructor, 651
  expired, 651
  lock, 651
  operator=, 651
  owner_before, 651
  reset, 651
  swap, 651
  use_count, 651
weakly_canonical, 1482
weakly_incrementable, 916
weekday, 1268
  c_encoding, 1269
  constructor, 1268, 1269
  from_stream, 1270
  iso_encoding, 1269
  ok, 1269
  operator+, 1269, 1270
  operator++, 1269
  operator=, 1269
  operator-, 1270
  operator--, 1269
  operator==, 1269
  operator<<, 1269
  operator[], 1269
  weekday_indexed, 1271
  weekday_last, 1272
  year_month_weekday, 1284
  year_month_weekday_last, 1286
weekday_indexed, 1270
  constructor, 1271
  index, 1271
  month_weekday, 1274
  ok, 1271
  operator<<, 1271
  operator=, 1271
  weekday, 1271
  year_month_weekday, 1284
weekday_last, 1271
  constructor, 1271
  month_weekday_last, 1275
  ok, 1272
  operator<<, 1272
  operator=, 1272
  weekday, 1272
  year_month_weekday_last, 1286
weeks, 1225
weibull_distribution, 1180
  a, 1180
  b, 1180
  constructor, 1180
  result_type, 1180
WEOF, 780, 781
wfilebuf, 1356, 1421
wformat_args, 721
wformat_context, 721, 734
wformat_parse_context, 721
wfstream, 1356, 1421
what
  bad_alloc, 510
  bad_any_cast, 604
  bad_array_new_length, 510
  bad_cast, 512
  bad_exception, 516
  bad_function_call, 682
  bad_optional_access, 500
  bad_typeid, 513
  bad_variant_access, 603
  bad_weak_ptr, 639
  exception, 516
  filesystem_error, 1458
  future_error, 1597
  system_error, 560
wide_string
  wstring_convert, 1679
widen
  basic_ios, 1371
  ctype, 1324
  ctype<char>, 1327
width
  ios_base, 1322, 1365, 1366
wifstream, 1356, 1421
wint_t, 780, 781
wios, 1360
wistream, 1356, 1383
wistringstream, 1356, 1407
wmemchr, 781
wmemcmp, 781
wmemcpyp, 781
wmemmove, 781
wmemset, 781
wofstream, 1356, 1421
wostream, 1356, 1383
wostringstream, 1356, 1407
wprintf, 781
wregex, 1488
write
  basic_ostream, 1401
ws, 1388, 1393
wscanf, 781
wstringbuf, 1356, 1375
wstreampos, 1356
wstring
  hash, 770
  operator"\", 770
  path, 1452
wstring_convert, 1678
  byte_string, 1679
  constructor, 1679
  destructor, 1680
  from_bytes, 1679
  int_type, 1679
  state, 1679
  state_type, 1679
  to_bytes, 1679
  wide_string, 1679
wstring_view
  hash, 779
  operator"sv", 779
wstringbuf, 1356, 1407
wstringstream, 1356, 1407
wsyncbuf, 1356, 1433
X
xalloc
  ios_base, 1366
xsgetn
  basic_streambuf, 1381
xsputn
  basic_streambuf, 1382
Y
year, 1265
  constructor, 1266
  from_stream, 1267
  is_leap, 1267
  max, 1267
  min, 1267
  ok, 1267
  operator"int", 1267
  operator"y", 1268
  operator+, 1267
  operator++, 1266
  operator+=, 1266
  operator-, 1267
  operator--, 1266
  operator-=, 1267
  operator<<, 1267
  operator<>, 1267
  operator==, 1267
  operator_local_days, 1279
  operator_sys_days, 1279
  operator+, 1280
  operator++, 1278, 1279
  operator+=, 1276
  operator-, 1277
  operator-=, 1276
  operator<, 1277
  operator<=, 1277
  operator==, 1276
  year, 1276
year_month_day, 1278
  constructor, 1278
  day, 1279
  from_stream, 1280
  month, 1279
  ok, 1279
  operator_local_days, 1279
  operator_sys_days, 1279
  operator+, 1280
  operator++, 1278, 1279
  operator+=, 1276
  operator-, 1277
  operator-=, 1276
  operator<, 1277
  operator<=, 1277
  operator==, 1276
  year, 1279
year_month_day_last, 1280
  constructor, 1281
  day, 1282
  month, 1281
  month_day_last, 1282
  ok, 1282
  operator_local_days, 1282
  operator_sys_days, 1282
  operator+, 1282
  operator++, 1281
  operator+=, 1281
  operator-, 1282
  operator-=, 1281
  operator<, 1283
  operator<=, 1282
  operator==, 1282
  year, 1281
year_month_weekday, 1283
  constructor, 1283
  index, 1284
  month, 1284
  ok, 1284
  operator_local_days, 1284
  operator_sys_days, 1284
  operator+, 1285
  operator++, 1284
  operator+=, 1284
  operator-, 1285
  operator-=, 1284
  operator<, 1285
  operator<=, 1284
  operator==, 1284
  weekday, 1284
  weekday_indexed, 1284
year, 1284
year_month_weekday_last, 1285
  constructor, 1286
month, 1286
  ok, 1286
operator local_days, 1286
operator sys_days, 1286
operator+, 1287
operator+=, 1286
operator-, 1287
operator=, 1286
operator<<, 1287
operator==, 1287
weekday, 1286
weekday_last, 1286
year, 1286
years, 1225
yield
  this_thread, 1565

Z
zero
  duration, 1243
  duration_values, 1240
zoned_time, 1300
  constructor, 1301–1303
  get_info, 1303
  get_local_time, 1303
  get_sys_time, 1303
  get_time_zone, 1303
  operator local_time, 1303
  operator sys_time, 1303
  operator<<, 1303
  operator=, 1303
  operator==, 1303
zoned_traits, 1299
zoned_traits<const time_zone*>&
  default_zone, 1300
  locate_zone, 1300
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**, 979, 980, 980, 981–984
- **assignable_from**, 536, 539, 539, 540, 545, 926, 927, 930, 943–945, 947, 949, 988
- **bidirectional_iterator**, 902, 905, 908, 919, 919, 920, 930–932, 940, 948, 950, 966, 967, 971, 975, 976, 1038, 1039, 1045, 1046, 1053, 1055, 1064, 1080, 1081, 1091, 1103, 1106, 1120, 1121
- **bidirectional_range**, 964, 971, 971, 973, 990, 991, 994–996, 1006–1008, 1017, 1020, 1021, 1038, 1039, 1045, 1046, 1053, 1055, 1064, 1080, 1081, 1091, 1104, 1107, 1120, 1121
- **boolean-testable**, 465, 524, 525, 542, 543, 543, 545
- **boolean-testable-impl**, 542, 542, 543
- **borrowed_range**, 897, 962, 969, 970, 974, 976, 977
- **can-reference**, 901, 901, 902, 911, 917, 993, 994, 997
- **common_range**, 971, 972, 1015
- **common_range_with**, 525, 536, 538, 538, 540
- **common_with**, 536, 538, 539, 948, 950, 951
- **compares-as**, 524, 525
- **constructible_from**, 536, 541, 541, 617, 618, 911, 926, 927, 945, 974, 978, 1010, 1011, 1015, 1016, 1558
- **contiguous_iterator**, 774, 778, 789, 896–899, 903, 908, 920, 920, 968, 1015, 1208
- **contiguous_range**, 897, 898, 971, 971, 989
- **convertible-to-non-slicing**, 973, 974–976
- **convertible_to**, 108, 384, 535, 538, 538, 542, 912, 943–945, 947, 949, 974, 976, 980, 989, 994, 996–998, 1000, 1002, 1006, 1008, 1009, 1011, 1012, 1015, 1020, 1022
- **copy_constructible**, 536, 542, 542, 545, 925, 926, 940, 962, 963, 978, 987, 987, 993, 994, 997, 1005, 1015, 1017, 1040, 1042, 1043, 1068, 1069, 1083, 1087
- **copyable**, 545, 911, 927, 943, 990, 991, 1006, 1015, 1060, 1061, 1115, 1116
- **cpp17-bidirectional_iterator**, 912, 912
- **cpp17-forward_iterator**, 911, 912
- **cpp17-input_iterator**, 911, 911, 912
- **cpp17-iterator**, 911, 911, 912
- **cpp17-random-access_iterator**, 912, 912
- **dereferenceable**, 901, 901, 902, 915, 944, 945, 948, 949
- **derived_from**, 535, 537, 537, 918–920, 932, 939, 945, 973, 974, 991, 995, 1007, 1014
- **destructor**, 463, 536, 541, 541, 619, 1557
- **equality_comparable**, 536, 544, 544, 911, 915, 917, 919, 995, 1007, 1020
- **equality_comparable_with**, 544, 544
- **equivalence_relation**, 537, 546, 546, 925
- **floating_point**, 539, 1224
- **has-arrow**, 972, 990, 991, 1006, 1008
- **has-tuple-element**, 1018, 1019
- **incrementable**, 902, 917, 917, 919, 980, 982, 983
- **indirect_binary_predicate**, 925, 927, 1035, 1033, 1041, 1043, 1044, 1069, 1072, 1073, 1084, 1085, 1087, 1088
- **indirect_equivalence_relation**, 925, 1035, 1036, 1044, 1045, 1076, 1089, 1090
- **indirect_strict_weak_order**, 926, 928, 1049–1052, 1055, 1060–1063, 1098–1102, 1107, 1114–1119
- **indirect_unary_predicate**, 925, 942, 963, 989, 990, 992, 1001, 1004, 1029–1031, 1033, 1038, 1041–1044, 1052, 1053, 1067, 1069, 1070, 1073, 1080, 1084, 1085, 1087, 1088, 1102–1105
- **indirectly-readable-impl**, 915
- **indirectly_readable**, 903, 926, 927, 927, 963, 1010–1013, 1032, 1034–1037, 1070, 1071, 1074, 1075, 1077, 1078
- **indirectly_copyable**, 903, 926, 927, 927, 928, 1037, 1038, 1041, 1042, 1044–1047, 1049, 1053, 1060, 1061, 1078–1080, 1085, 1088, 1090–1093, 1098, 1104, 1115
- **indirectly_copyable_storable**, 927, 927, 1045, 1090, 1116, 1117
view, 963, 964, 969, 970, 970, 971–973, 977, 978, 987–989, 992, 998–1002, 1004, 1009, 1014–1017, 1019
viewable_range, 962, 972, 972, 987, 1015

weakly-equality-comparable-with, 525, 543, 544, 918, 962, 979, 982, 984
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, 455

additional execution policies supported by parallel algorithms, 718, 1028
additional file_type enumerators for file systems supporting additional types of file, 1459
additional formats for time_get::do_get_date, 1344
additional supported forms of preprocessing directive, 443
algorithms for producing the standard random number distributions, 1173
alignment, 64
alignment additional values, 65
alignment of bit-fields within a class object, 270
allocation of bit-fields within a class object, 270
any use of an invalid pointer other than to perform indirection or deallocate, 61
argument values to construct ios_base::failure, 1372
assignability of placeholder objects, 682
behavior of iostream classes when traits::pos_type is not streampos or when traits::off_type is not streamoff, 1356
behavior of non-standard attributes, 230
behavior of strstreambuf::setbuf, 1669
bits in a byte, 54
choice of larger or smaller value of floating-point-literal, 21
concatenation of some types of string-literals, 22
conversions between pointers and integers, 118
converting characters from source character set to execution character set, 10
converting function pointer to object pointer and vice versa, 118
default configuration of a pool, 660
default next_buffer_size for a monotonic_buffer_resource, 661
default number of buckets in unordered_map, 871
default number of buckets in unordered_multimap, 877
default number of buckets in unordered_multiset, 886
default number of buckets in unordered_set, 881
defining main in freestanding environment, 82
definition and meaning of __STDC_VERSION__, 457
definition of NULL, 490, 1659
derived type for typeid, 115
diagnostic message, 3
dynamic initialization of static inline variables before main, 84
dynamic initialization of static variables before main, 83, 84
dynamic initialization of thread-local variables before entry, 84
effect of calling associated Laguerre polynomials with n >= 128 or m >= 128, 1219
effect of calling associated Legendre polynomials with l >= 128, 1219
effect of calling basic_filebuf::setbuf with nonzero arguments, 1426
effect of calling basic_filebuf::sync when a get area exists, 1426
effect of calling basic_streambuf::setbuf with nonzero arguments, 1413
effect of calling cylindrical Bessel functions of the first kind with nu >= 128, 1220
effect of calling cylindrical Neumann functions with nu >= 128, 1221
effect of calling Hermite polynomials with n >= 128, 1222
effect of calling io_base::sync_with_stdio after any input or output operation on standard streams, 1366
effect of calling irregular modified cylindrical Bessel functions with nu >= 128, 1220
effect of calling Laguerre polynomials with n >= 128, 1222
effect of calling Legendre polynomials with l >= 128, 1222
effect of calling regular modified cylindrical Bessel functions with nu >= 128, 1220
effect of calling spherical associated Legendre functions with l >= 128, 1223
effect of calling spherical Bessel functions with n >= 128, 1223
effect of calling spherical Neumann functions with n >= 128, 1223
effect of filesystem::copy, 1471
effect on C locale of calling locale::global, 1321
encoding of universal character name not in execution character set, 20
error_category for errors originating outside the operating system, 487
exception type when `random_device` constructor fails, 1171
exception type when `random_device::operator()` fails, 1171
exception type when `shared_ptr` constructor fails, 642
exceptions thrown by standard library functions that have a potentially-throwing exception specification, 487
execution character set and execution wide-character set, 12
exit status, 504
extended signed integer types, 70

file type of the file argument of `filesystem::status`, 1481
formatted character sequence generated by `time_put::do_put` in C locale, 1346
forward progress guarantees for implicit threads of parallel algorithms (if not defined for `thread`), 1027
growth factor for `monotonic_buffer_resource`, 661, 662
headers for freestanding implementation, 470
how `random_device::operator()` generates values, 1171
how the set of importable headers is determined, 240
interactive device, 8
interpretation of the path character sequence with format `path::auto_format`, 1458
largest supported value to configure the largest allocation satisfied directly by a pool, 659
largest supported value to configure the maximum number of blocks to replenish a pool, 659
linkage of `main`, 82
linkage of names from C standard library, 471
linkage of objects between C++ and other languages, 228
locale names, 1320
lvalue-to-rvalue conversion of an invalid pointer value, 91
manner of search for included source file, 446
mapping from name to catalog when calling `messages::do_open`, 1353
mapping from physical source file characters to basic source character set, 11, 1640
mapping header name to header or external source file, 14
mapping of pointer to integer, 118
mapping physical source file characters to basic source character set, 10
mapping to message when calling `messages::do_get`, 1353
maximum depth of recursive template instantiations, 406
maximum size of an allocated object, 125, 510
meaning of `,`, `\`, `/`, `*`, or `/` in a q-char-sequence or an h-char-sequence, 14
meaning of `asm` declaration, 226
meaning of attribute declaration, 157
meaning of dot-dot in `root-directory`, 1446
negative value of `character-literal` in preprocessor, 445
nesting limit for `#include` directives, 447
NTCTS in `basic_ostream<charT, traits>& operator<<(nullptr_t)`, 1400
number of placeholders for bind expressions, 667, 682
number of threads in a program under a freestanding implementation, 77
numeric values of `character-literals` in `#if` directives, 445
operating system on which implementation depends, 1438
parameters to `main`, 82
passing argument of class type through ellipsis, 113
physical source file characters, 10
presence and meaning of `native_handle_type` and `native_handle`, 1551
range defined for `character-literals`, 19
rank of extended signed integer type, 74
required alignment for `atomic_ref` type’s operations, 1527, 1530–1532
required libraries for freestanding implementation, 7
resource limits on a message catalog, 1353
result of `filesystem::file_size`, 1476
result of inexact floating-point conversion, 93
return value of `bad_alloc::what`, 510
return value of `bad_any_cast::what`, 604
return value of `bad_array_new_length::what`, 510
return value of `bad_cast::what`, 512
return value of `bad_exception::what`, 516
return value of `bad_function_call::what`, 682
return value of `bad_optional_access::what`, 590
return value of `bad_typeid::what`, 513
return value of `bad_variant_access::what`, 603
return value of `bad_weak_ptr::what`, 639
return value of `char_traits<char16_t>::eof`, 743
return value of `char_traits<char32_t>::eof`, 743
return value of `char_traits<uchar_t>::eof`, 743
return value of `exception::what`, 516
return value of `type_info::name()`, 512
search locations for `"` header, 447

Index of impl.-def. behavior 1824
search locations for <> header, 446
semantics of an access through a volatile glvalue, 167
semantics of linkage specification on templates, 347
semantics of linkage specifiers, 226
semantics of non-standard escape sequences, 19
semantics of parallel algorithms invoked with implementation-defined execution policies, 1028
semantics of token parameter and default token value used by random_device constructors, 1171
sequence of places searched for a header, 446
set of character types that iostreams templates can be instantiated for, 1319, 1356
signedness of char, 168
sizeof applied to fundamental types other than char, signed char, and unsigned char, 123
stack unwinding before call to std::terminate, 437, 440
startup and termination in freestanding environment, 82
strict total order over pointer values, 460
string resulting from __func__, 205
support for always lock-free integral atomic types in freestanding environments, 470
support for extended alignments, 65
support for module-import-declarations with non-C++ language linkage, 227
supported multibyte character encoding rules, 742
supported root-names in addition to any operating system dependent root-names, 1445, 1446
text of __DATE__ when date of translation is not available, 455
text of __TIME__ when time of translation is not available, 455
threads and program points at which deferred dynamic initialization is performed, 83, 84
type aliases atomic_signed_lock_free and atomic_unsigned_lock_free in freestanding environments, 470
type of a directory-like file, 1465, 1467
type of array::const_iterator, 822
type of array::iterator, 822
type of basic_string::const_iterator, 748
type of basic_string::iterator, 748
type of basic_string_view::const_iterator, 771, 774
type of default_random_engine, 1170
type of deque::const_iterator, 824
type of deque::iterator, 824
type of forward_list::const_iterator, 828
type of forward_list::iterator, 828
type of list::const_iterator, 834
type of list::iterator, 834
type of map::const_iterator, 849
type of map::iterator, 849
type of multimap::const_iterator, 854
type of multimap::iterator, 854
type of multiset::const_iterator, 861
type of multiset::iterator, 861
type of ptdiff_t, 132, 491
type of regex_constants::error_type, 1494
type of regex_constants::match_flag_type, 1493
type of set::const_iterator, 858
type of set::iterator, 858
type of size_t, 491
type of span::iterator, 895, 899
type of syntax_option_type, 1492
type of unordered_map::const_iterator, 867
type of unordered_map::const_local_iterator, 867
type of unordered_map::iterator, 873
type of unordered_multimap::const_iterator, 873
type of unordered_multimap::const_local_iterator, 873
type of unordered_multimap::iterator, 873
type of unordered_multiset::const_iterator, 882
type of unordered_multiset::const_local_iterator, 882
type of unordered_multiset::const_local_iterator, 882
type of unordered_multiset::iterator, 882
type of unordered_set::const_iterator, 878
type of unordered_set::const_local_iterator, 878
type of unordered_set::iterator, 878
type of unordered_set::local_iterator, 878
type of vector::const_iterator, 840
type of vector::iterator, 840
type of vector<bool>::const_iterator, 844
type of vector<bool>::iterator, 844
underlying type for enumeration, 212
underlying type of bool, 71
underlying type of char, 71
underlying type of wchar_t, 71
unit suffix when Period::type is micro, 1247
value for least_max_value default template argument of counting_semaphore, 1591
value of bit-field that cannot represent assigned value, 139
incremented value, 114
initializer, 192
value of character-literal outside range of corresponding type, 19
value of ctype<char>::table_size, 1326
value of has-attribute-expression for non-standard attributes, 445
value of multicharacter literal, 19
value of pow(0,0), 1149
value of result of inexact integer to floating-point conversion, 93
value of wide-character literal containing multiple characters, 19
value of wide-character literal with single c-char that is not in execution wide-character set, 19
value representation of floating-point types, 71
value representation of pointer types, 72
values of a trivially copyable type, 69
values of various ATOMIC_..._LOCK_FREE macros, 1526
whether <cfenv> functions can be used to manage floating-point status, 1142
whether a given atomic type’s operations are always lock free, 1533, 1535, 1538, 1540, 1541, 1544, 1546
whether a given atomic_ref type’s operations are always lock free, 1527, 1530–1532
whether an implementation has relaxed or strict pointer safety, 64
whether functions from Annex K of the C standard library are declared when C++ headers are included, 469
whether get_pointer_safety returns pointer_safety::relaxed or pointer_safety::preferred if the implementation has relaxed pointer safety, 624
whether locale object is global or per-thread, 1318
whether pragma FENV_ACCESS is supported, 1142
whether rand may introduce a data race, 1190
whether sequence pointers are copied by basic_filebuf move constructor, 1423
whether sequence pointers are copied by basic_stringbuf move constructor, 1409
whether sequence pointers are initialized to null pointers, 1409
whether source file inclusion of importable header is replaced with import directive, 447
whether source of translation units must be available to locate template definitions, 11
whether stack is unwound before calling the function std::terminate when a noexcept specification is violated, 440
whether the implementation is hosted or freestanding, 470
whether the lifetime of a parameter ends when the callee returns or at the end of the enclosing full-expression, 112
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, 11
whether the thread that executes main and the threads created by std::thread or std::jthread provide concurrent forward progress guarantees, 81
whether time_get::do_get_year accepts two-digit year numbers, 1345
whether values are rounded or truncated to the required precision when converting between time_t values and time_point objects, 1251
which functions in the C++ standard library may be recursively reentered, 486
which non-standard-layout objects containing no data are considered empty, 56
which scalar types have unique object representations, 703
width of integral type, 71