Working Draft, Standard for Programming Language C++

Note: this is an early draft. It’s known to be incomple 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** ............................................................... 6
   4.1 Implementation compliance ..................................................... 6
   4.2 Structure of this document ..................................................... 7
   4.3 Syntax notation .................................................................... 7
   4.4 Acknowledgments .................................................................. 8

5 **Lexical conventions** ............................................................. 9
   5.1 Separate translation ............................................................... 9
   5.2 Phases of translation ............................................................. 9
   5.3 Character sets ..................................................................... 10
   5.4 Preprocessing tokens ........................................................... 11
   5.5 Alternative tokens ............................................................... 12
   5.6 Tokens .............................................................................. 12
   5.7 Comments ......................................................................... 12
   5.8 Header names ..................................................................... 12
   5.9 Preprocessing numbers ....................................................... 13
   5.10 Identifiers ......................................................................... 13
   5.11 Keywords ......................................................................... 14
   5.12 Operators and punctuators ................................................... 14
   5.13 Literals ............................................................................. 15

6 **Basics** .................................................................................. 24
   6.1 Declarations and definitions ................................................... 24
   6.2 One-definition rule .............................................................. 26
   6.3 Scope .............................................................................. 30
   6.4 Name lookup ...................................................................... 35
   6.5 Program and linkage ............................................................ 47
   6.6 Memory and objects ............................................................ 50
   6.7 Types ............................................................................... 63
   6.8 Program execution ............................................................. 69

7 **Expressions** .......................................................................... 81
   7.1 Preamble ........................................................................... 81
   7.2 Properties of expressions ..................................................... 82
   7.3 Standard conversions .......................................................... 85
   7.4 Usual arithmetic conversions ................................................ 89
   7.5 Primary expressions ............................................................ 90
   7.6 Compound expressions ........................................................ 105
   7.7 Constant expressions .......................................................... 135

8 **Statements** .......................................................................... 141
   8.1 Labeled statement .............................................................. 142
   8.2 Expression statement ........................................................... 142
   8.3 Compound statement or block ............................................. 142
   8.4 Selection statements ............................................................ 142
   8.5 Iteration statements ............................................................. 144
   8.6 Jump statements ................................................................ 147
   8.7 Declaration statement .......................................................... 148
14.3 Handling an exception .................................................. 421
14.4 Exception specifications .................................................. 422
14.5 Special functions .......................................................... 425

15 Preprocessing directives .................................................. 427
15.1 Conditional inclusion ..................................................... 428
15.2 Source file inclusion ....................................................... 431
15.3 Global module fragment .................................................. 432
15.4 Header units ............................................................... 432
15.5 Macro replacement ......................................................... 433
15.6 Line control ............................................................... 438
15.7 Error directive ............................................................. 439
15.8 Pragma directive .......................................................... 439
15.9 Null directive ............................................................. 439
15.10 Predefined macro names ............................................... 439
15.11 Pragma operator .......................................................... 441

16 Library introduction ....................................................... 443
16.1 General ................................................................. 443
16.2 The C standard library .................................................. 444
16.3 Definitions ............................................................... 444
16.4 Method of description .................................................. 447
16.5 Library-wide requirements ............................................. 453

17 Language support library ................................................. 473
17.1 General ................................................................. 473
17.2 Common definitions ..................................................... 473
17.3 Implementation properties ............................................. 477
17.4 Integer types ............................................................ 487
17.5 Start and termination ................................................... 488
17.6 Dynamic memory management ...................................... 489
17.7 Type identification ..................................................... 496
17.8 Contract violation handling .......................................... 498
17.9 Exception handling ..................................................... 499
17.10 Initializer lists .......................................................... 502
17.11 Comparisons ........................................................... 503
17.12 Coroutines ............................................................. 511
17.13 Other runtime support ................................................. 515

18 Concepts library .......................................................... 518
18.1 General ................................................................. 518
18.2 Equality preservation ................................................... 518
18.3 Header <concepts> synopsis ......................................... 519
18.4 Language-related concepts .......................................... 521
18.5 Comparison concepts .................................................. 526
18.6 Object concepts ........................................................ 528
18.7 Callable concepts ....................................................... 528

19 Diagnostics library ......................................................... 530
19.1 General ................................................................. 530
19.2 Exception classes ....................................................... 530
19.3 Assertions .............................................................. 533
19.4 Error numbers .......................................................... 533
19.5 System error support ................................................... 535

20 General utilities library .................................................. 544
20.1 General ................................................................. 544
20.2 Utility components ..................................................... 544
20.3 Compile-time integer sequences ................................... 548
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.4</td>
<td>548</td>
</tr>
<tr>
<td>21.5</td>
<td>553</td>
</tr>
<tr>
<td>21.6</td>
<td>562</td>
</tr>
<tr>
<td>21.7</td>
<td>575</td>
</tr>
<tr>
<td>21.8</td>
<td>586</td>
</tr>
<tr>
<td>21.9</td>
<td>591</td>
</tr>
<tr>
<td>21.10</td>
<td>597</td>
</tr>
<tr>
<td>21.11</td>
<td>618</td>
</tr>
<tr>
<td>21.12</td>
<td>645</td>
</tr>
<tr>
<td>21.13</td>
<td>654</td>
</tr>
<tr>
<td>21.14</td>
<td>658</td>
</tr>
<tr>
<td>21.15</td>
<td>681</td>
</tr>
<tr>
<td>21.16</td>
<td>705</td>
</tr>
<tr>
<td>21.17</td>
<td>708</td>
</tr>
<tr>
<td>21.18</td>
<td>709</td>
</tr>
<tr>
<td>21.19</td>
<td>711</td>
</tr>
</tbody>
</table>

### 21 Strings library

- 21.1 General 714
- 21.2 Character traits 714
- 21.3 String classes 719
- 21.4 String view classes 746
- 21.5 Null-terminated sequence utilities 755

### 22 Containers library

- 22.1 General 760
- 22.2 Container requirements 760
- 22.3 Sequence containers 796
- 22.4 Associative containers 823
- 22.5 Unordered associative containers 841
- 22.6 Container adaptors 864
- 22.7 Views 872

### 23 Iterators library

- 23.1 General 878
- 23.2 Header `<iterator>` synopsis 878
- 23.3 Iterator requirements 884
- 23.4 Iterator primitives 903
- 23.5 Iterator adaptors 906
- 23.6 Stream iterators 928
- 23.7 Range access 934

### 24 Ranges library

- 24.1 General 936
- 24.2 Header `<ranges>` synopsis 936
- 24.3 Range access 939
- 24.4 Range requirements 942
- 24.5 Range utilities 945
- 24.6 Range factories 951
- 24.7 Range adaptors 957

### 25 Algorithms library

- 25.1 General 986
- 25.2 Algorithms requirements 986
- 25.3 Parallel algorithms 988
- 25.4 Header `<algorithm>` synopsis 991
- 25.5 Non-modifying sequence operations 1027
- 25.6 Mutating sequence operations 1038
- 25.7 Sorting and related operations 1054

---

© ISO/IEC N4810
26 Numerics library .............................................................. 1094
26.1 General ............................................................... 1094
26.2 Numeric type requirements ........................................ 1094
26.3 The floating-point environment .................................... 1094
26.4 Complex numbers .................................................... 1095
26.5 Bit manipulation .................................................... 1103
26.6 Random number generation ........................................ 1105
26.7 Numeric arrays ..................................................... 1142
26.8 Mathematical functions for floating-point types .............. 1161
27 Time library .................................................................. 1177
27.1 General ............................................................... 1177
27.2 Header <chrono> synopsis ........................................... 1177
27.3 C++17Clock requirements ........................................... 1192
27.4 Time-related traits .................................................... 1193
27.5 Class template duration .............................................. 1194
27.6 Class template time_point ............................................ 1201
27.7 Clocks ................................................................ 1204
27.8 The civil calendar ...................................................... 1216
27.9 Class template time_of_day ......................................... 1245
27.10 Time zones ........................................................... 1250
27.11 Formatting ............................................................ 1263
27.12 Parsing ................................................................ 1266
27.13 Header <ctime> synopsis ........................................... 1269
28 Localization library .......................................................... 1271
28.1 General ............................................................... 1271
28.2 Header <locale> synopsis ........................................... 1271
28.3 Locales ................................................................. 1272
28.4 Standard locale categories ......................................... 1279
28.5 C library locales ...................................................... 1309
29 Input/output library .......................................................... 1311
29.1 General ............................................................... 1311
29.2 Iostreams requirements .............................................. 1311
29.3 Forward declarations ............................................... 1312
29.4 Standard iostream objects .......................................... 1314
29.5 Iostreams base classes .............................................. 1315
29.6 Stream buffers ....................................................... 1331
29.7 Formatting and manipulators ...................................... 1339
29.8 String-based streams ............................................... 1362
29.9 File-based streams .................................................. 1371
29.10 Synchronized output streams ..................................... 1384
29.11 File systems ........................................................ 1389
29.12 C library files ...................................................... 1434
30 Regular expressions library .............................................. 1438
30.1 General ............................................................... 1438
30.2 Definitions ............................................................ 1438
30.3 Requirements ........................................................ 1439
30.4 Header <regex> synopsis ........................................... 1440
30.5 Namespace std::regex_constants ................................ 1446
30.6 Class regex_error .................................................... 1449
30.7 Class template regex_traits ....................................... 1449
30.8 Class template basic_regex ........................................ 1451
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.9</td>
<td>Class template <code>sub_match</code></td>
<td>1456</td>
</tr>
<tr>
<td>30.10</td>
<td>Class template <code>match_results</code></td>
<td>1460</td>
</tr>
<tr>
<td>30.11</td>
<td>Regular expression algorithms</td>
<td>1465</td>
</tr>
<tr>
<td>30.12</td>
<td>Regular expression iterators</td>
<td>1469</td>
</tr>
<tr>
<td>30.13</td>
<td>Modified ECMAScript regular expression grammar</td>
<td>1474</td>
</tr>
<tr>
<td>31</td>
<td>Atomic operations library</td>
<td>1477</td>
</tr>
<tr>
<td>31.1</td>
<td>General</td>
<td>1477</td>
</tr>
<tr>
<td>31.2</td>
<td>Header <code>&lt;atomic&gt;</code> synopsis</td>
<td>1477</td>
</tr>
<tr>
<td>31.3</td>
<td>Type aliases</td>
<td>1481</td>
</tr>
<tr>
<td>31.4</td>
<td>Order and consistency</td>
<td>1481</td>
</tr>
<tr>
<td>31.5</td>
<td>Lock-free property</td>
<td>1482</td>
</tr>
<tr>
<td>31.6</td>
<td>Class template <code>atomic_ref</code></td>
<td>1483</td>
</tr>
<tr>
<td>31.7</td>
<td>Class template <code>atomic</code></td>
<td>1488</td>
</tr>
<tr>
<td>31.8</td>
<td>Non-member functions</td>
<td>1497</td>
</tr>
<tr>
<td>31.9</td>
<td>Flag type and operations</td>
<td>1497</td>
</tr>
<tr>
<td>31.10</td>
<td>Fences</td>
<td>1498</td>
</tr>
<tr>
<td>32</td>
<td>Thread support library</td>
<td>1500</td>
</tr>
<tr>
<td>32.1</td>
<td>General</td>
<td>1500</td>
</tr>
<tr>
<td>32.2</td>
<td>Requirements</td>
<td>1500</td>
</tr>
<tr>
<td>32.3</td>
<td>Threads</td>
<td>1502</td>
</tr>
<tr>
<td>32.4</td>
<td>Mutual exclusion</td>
<td>1507</td>
</tr>
<tr>
<td>32.5</td>
<td>Condition variables</td>
<td>1525</td>
</tr>
<tr>
<td>32.6</td>
<td>Futures</td>
<td>1531</td>
</tr>
<tr>
<td>A</td>
<td>Grammar summary</td>
<td>1545</td>
</tr>
<tr>
<td>A.1</td>
<td>Keywords</td>
<td>1545</td>
</tr>
<tr>
<td>A.2</td>
<td>Lexical conventions</td>
<td>1545</td>
</tr>
<tr>
<td>A.3</td>
<td>Basics</td>
<td>1549</td>
</tr>
<tr>
<td>A.4</td>
<td>Expressions</td>
<td>1549</td>
</tr>
<tr>
<td>A.5</td>
<td>Statements</td>
<td>1553</td>
</tr>
<tr>
<td>A.6</td>
<td>Declarations</td>
<td>1554</td>
</tr>
<tr>
<td>A.7</td>
<td>Modules</td>
<td>1560</td>
</tr>
<tr>
<td>A.8</td>
<td>Classes</td>
<td>1560</td>
</tr>
<tr>
<td>A.9</td>
<td>Overloading</td>
<td>1562</td>
</tr>
<tr>
<td>A.10</td>
<td>Templates</td>
<td>1562</td>
</tr>
<tr>
<td>A.11</td>
<td>Exception handling</td>
<td>1563</td>
</tr>
<tr>
<td>A.12</td>
<td>Preprocessing directives</td>
<td>1564</td>
</tr>
<tr>
<td>B</td>
<td>Implementation quantities</td>
<td>1566</td>
</tr>
<tr>
<td>C</td>
<td>Compatibility</td>
<td>1568</td>
</tr>
<tr>
<td>C.1</td>
<td>C++ and ISO C</td>
<td>1568</td>
</tr>
<tr>
<td>C.2</td>
<td>C++ and ISO C++ 2003</td>
<td>1576</td>
</tr>
<tr>
<td>C.3</td>
<td>C++ and ISO C++ 2011</td>
<td>1581</td>
</tr>
<tr>
<td>C.4</td>
<td>C++ and ISO C++ 2014</td>
<td>1583</td>
</tr>
<tr>
<td>C.5</td>
<td>C++ and ISO C++ 2017</td>
<td>1586</td>
</tr>
<tr>
<td>C.6</td>
<td>C standard library</td>
<td>1591</td>
</tr>
<tr>
<td>D</td>
<td>Compatibility features</td>
<td>1594</td>
</tr>
<tr>
<td>D.1</td>
<td>Arithmetic conversion on enumerations</td>
<td>1594</td>
</tr>
<tr>
<td>D.2</td>
<td>Implicit capture of <code>*this</code> by reference</td>
<td>1594</td>
</tr>
<tr>
<td>D.3</td>
<td>Array comparisons</td>
<td>1594</td>
</tr>
<tr>
<td>D.4</td>
<td>Redeclaration of <code>static constexpr</code> data members</td>
<td>1594</td>
</tr>
<tr>
<td>D.5</td>
<td>Implicit declaration of copy functions</td>
<td>1594</td>
</tr>
<tr>
<td>D.6</td>
<td>C standard library headers</td>
<td>1595</td>
</tr>
<tr>
<td>D.7</td>
<td>Relational operators</td>
<td>1595</td>
</tr>
<tr>
<td>D.8</td>
<td><code>char*</code> streams</td>
<td>1596</td>
</tr>
</tbody>
</table>
D.9 Deprecated type traits .................................................. 1604
D.10 Deprecated iterator primitives ....................................... 1604
D.11 Deprecated move_iterator access .................................... 1605
D.12 Deprecated shared_ptr atomic access ............................... 1605
D.13 Deprecated basic_string capacity .................................... 1607
D.14 Deprecated standard code conversion facets ....................... 1607
D.15 Deprecated convenience conversion interfaces .................... 1608
D.16 Deprecated locale category facets ................................. 1612
D.17 Deprecated filesystem path factory functions .................... 1612

Bibliography ..................................................................... 1614

Cross references .................................................................. 1615

Cross references from ISO C++ 2017 .................................... 1637

Index ............................................................................. 1640

Index of grammar productions ........................................... 1672

Index of library headers ..................................................... 1677

Index of library names ........................................................ 1679

Index of implementation-defined behavior ............................. 1744
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 9899:2011 Programming languages — C (hereinafter referred to as the C standard). C++ provides many facilities beyond those provided by C, including additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.
2 Normative references

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


(1.3) — ISO/IEC 2382 (all parts), *Information technology — Vocabulary*

(1.4) — ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*

(1.5) — ISO/IEC 9899:2011, *Programming languages — C*

(1.6) — ISO/IEC 9945:2003, *Information Technology — Portable Operating System Interface (POSIX)*

(1.7) — ISO/IEC 10646, *Information technology — Universal Coded Character Set (UCS)*


(1.10) — ISO 80000-2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*

The library described in Clause 7 of ISO/IEC 9899:2011 is hereinafter called the *C standard library*.\(^1\)

The operating system interface described in ISO/IEC 9945:2003 is hereinafter called *POSIX*.\(^3\)

The ECMA Script Language Specification described in Standard Ecma-262 is hereinafter called *ECMA-262*.\(^4\)

*Note: References to ISO/IEC 10646-1:1993 are used only to support deprecated features (D.14). — 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:
   — ISO Online browsing platform: available at https://www.iso.org/obp

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 access [defns.access]
(execution-time action) read or modify the value of an object

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

3.3 argument [defns.argument.macro]
(function-like macro) sequence of preprocessing tokens in the comma-separated list bounded by the parentheses (15.5)

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

3.5 argument [defns.argument.templ]
(template instantiation) constant-expression, type-id, or id-expression in the comma-separated list bounded by the angle brackets (13.3)

3.6 block [defns.block]
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 conditionally-supported [defns.cond.supp]
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]

3.8 diagnostic message [defns.diagnostic]
message belonging to an implementation-defined subset of the implementation’s output messages

3.9 dynamic type [defns.dynamic.type]
(glvalue) type of the most derived object (6.6.2) to which the glvalue refers

[Example: If a pointer (9.2.3.1) p whose static type is “pointer to class B” is pointing to an object of class D, derived from B (11.6), the dynamic type of the expression *p is “D”. References (9.2.3.2) are treated similarly. — end example]
3.10 [defns.dynamic.type.prvalue]
dynamic type
(\text{prvalue}) static type of the \text{prvalue} expression

3.11 [defns.ill.formed]
il-formed program
program that is not well-formed (3.29)

3.12 [defns.impl.defined]
implementation-defined behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

3.13 [defns.impl.limits]
implementation limits
restrictions imposed upon programs by the implementation

3.14 [defns.locale.specific]
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

3.15 [defns.multibyte]
multibyte character
sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

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

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

3.17 [defns.parameter.macro]
parameter
(function-like macro) identifier from the comma-separated list bounded by the parentheses immediately following the macro name

3.18 [defns.parameter.templ]
parameter
(template) member of a template-parameter-list

3.19 [defns.signature]
signature
(function) name, parameter type list (9.2.3.5), enclosing namespace (if any), and trailing \textit{requires-clause} (9.2) (if any)

\begin{quote}
\textbf{[Note 1 to entry]:} Signatures are used as a basis for name mangling and linking. — end note
\end{quote}

3.20 [defns.signature.templ]
signature
(function template) name, parameter type list (9.2.3.5), enclosing namespace (if any), return type, template-head, and trailing \textit{requires-clause} (9.2) (if any)

3.21 [defns.signature.spec]
signature
(function template specialization) signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced)
3.22 signature
(class member function) name, parameter type list (9.2.3.5), class of which the function is a member, 
.cv-qualifiers (if any), ref-qualifier (if any), and trailing requires-clause (9.2) (if any)

3.23 signature
(class member function template) name, parameter type list (9.2.3.5), class of which the function is a member, 
.cv-qualifiers (if any), ref-qualifier (if any), return type (if any), template-head, and trailing requires-clause (9.2) 
(if any)

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

3.25 static type
type of an expression (6.7) resulting from analysis of the program without considering execution semantics

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

3.26 unblock
satisfy a condition that one or more blocked threads of execution are waiting for

3.27 undefined behavior
behavior for which this document imposes no requirements

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

3.28 unspecified behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation

[Note 1 to entry: The implementation is not required to document which behavior occurs. The range of 
possible behaviors is usually delineated by this document. — end note]

3.29 well-formed program
C++ program constructed according to the syntax rules, diagnosable semantic rules, and the one-definition 
rule (6.2)
4 General principles [intro]

4.1 Implementation compliance [intro.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 that program.

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

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

[Note: During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 13.9.2. — end note]

3 For classes and class templates, the library Clauses specify partial definitions. Private members (11.8) 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 (9.7). A C++ translation unit (5.2) obtains access to these names by including the appropriate standard library header (15.2).

6 The templates, classes, functions, and objects in the library have external linkage (6.5). 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.

4.1.1 Abstract machine [intro.abstract]

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.

---

2) “Correct execution” can include undefined behavior, depending on the data being processed; see Clause 3 and 6.8.1.

3) This documentation also defines implementation-defined behavior; see 6.8.1.

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.
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). 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: ” 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

```
{ expression\textsubscript{opt} }
```

indicates an optional expression enclosed in braces.

5) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 4.1.
Names for syntactic categories have generally been chosen according to the following rules:

(2.1) — X-name is a use of an identifier in a context that determines its meaning (e.g., class-name, typedef-name).

(2.2) — X-id is an identifier with no context-dependent meaning (e.g., qualified-id).

(2.3) — X-seq is one or more X’s without intervening delimiters (e.g., declaration-seq is a sequence of declarations).

(2.4) — X-list is one or more X’s separated by intervening commas (e.g., identifier-list is a sequence of identifiers separated by commas).

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.

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 (15.2) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (15.1) 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 (6.5) 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 (6.5). —end note]

5.2 Phases of translation

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

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. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty 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 (15.5.3), 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 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.

---

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.

§ 5.2
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 (13.2). — 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 (13.8.2). — 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  

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

```
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  0 1 2 3 4 5 6 7 8 9
  _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | 
```

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

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

The character designated by the universal-character-name \U00NNNNNNN is that character that has U+NNNNNNN as a code point short identifier; the character designated by the universal-character-name \uNNNNN is that character that has U+NNNNN as a code point short identifier. If a universal-character-name does not correspond to a code point in ISO/IEC 10646 or if a universal-character-name corresponds to a surrogate code point, the program is ill-formed. Additionally, if a universal-character-name outside the c-char-sequence, s-char-sequence, or r-char-sequence of a character or string literal corresponds to a control character or to a character in the basic source character set, the program is ill-formed. [Note: ISO/IEC 10646 code points are within the range 0x0-0x10FFFF (inclusive). A surrogate code point is a value in the range 0xD800-0xDFFF (inclusive). A control character is a character whose code point is in either of the ranges 0x0-0x1F or 0x7F-0x9F (both inclusive). — 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

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

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, 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.

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

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

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. 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

   — within a #include directive (15.2),
   — within a has-include-expression, or
   — outside of any preprocessing directive, if applying phase 4 of translation to the sequence of preprocessing tokens produced thus far is valid and results in an import-seq (15.4).

[Example:

```c
#define R "x"
const char* s = R"y"; // ill-formed raw string, not "x" "y"
```

— end example]

4. [Example: The program fragment 0xe+foo is parsed as a preprocessing number token (one that is not a valid floating or integer 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 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.** Alternative token representations are provided for some operators and punctuators.**

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

### Table 1 — Alternative tokens

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Primary</th>
<th>Alternative</th>
<th>Primary</th>
<th>Alternative</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; %</td>
<td>and</td>
<td>%</td>
<td>and_eq</td>
<td>% =</td>
<td></td>
</tr>
<tr>
<td>%&gt;</td>
<td>}</td>
<td>bitor</td>
<td>or_eq</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>&lt;:</td>
<td>[</td>
<td>or</td>
<td>xor_eq</td>
<td></td>
<td>=</td>
</tr>
<tr>
<td>:&gt;</td>
<td>]</td>
<td>xor</td>
<td>not</td>
<td>!</td>
<td></td>
</tr>
<tr>
<td>%: %:</td>
<td>#</td>
<td>compl</td>
<td>not_eq</td>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>%: %:</td>
<td>#</td>
<td>bitand</td>
<td>&amp;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.6 Tokens

*token:* identifier keyword literal operator punctuator

There are five kinds of tokens: identifiers, keywords, literals, 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]

5.8 Header names

*header-name:* `< h-char-sequence >`

`* q-char-sequence `*

h-char-sequence:

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

---

**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 (15.5.2) 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
h-char:

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

q-char-sequence:

q-char
q-char-sequence q-char

q-char:

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

1 [Note: Header name preprocessing tokens only appear within a #include preprocessing directive (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 15.2.

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.14

5.9 Preprocessing numbers [lex.pppnumber]

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 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 literal token.

5.10 Identifiers [lex.name]

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

1 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.15

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.

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...
Table 2 — Ranges of characters allowed

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00A8</td>
<td>00AA</td>
<td>00AD</td>
<td>00AF</td>
<td>00B2–00B5</td>
</tr>
<tr>
<td>00B7–00BA</td>
<td>00B8–00BE</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–FDCE</td>
<td>FDF0–FE44</td>
<td>FE47–FFFD</td>
<td></td>
</tr>
<tr>
<td>10000–1FFFF</td>
<td>20000–2FFFF</td>
<td>30000–3FFFF</td>
<td>40000–4FFFF</td>
<td>50000–5FFFF</td>
</tr>
<tr>
<td>60000–6FFFF</td>
<td>70000–7FFFF</td>
<td>80000–8FFFF</td>
<td>90000–9FFFF</td>
<td>A0000–AFFFD</td>
</tr>
<tr>
<td>B0000–BFFFF</td>
<td>C0000–CFFFF</td>
<td>D0000–DFFFF</td>
<td>E0000–EFFFF</td>
<td></td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0300–036F</td>
<td>1DC0–1DFF</td>
<td>20D0–20FF</td>
</tr>
</tbody>
</table>

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

Table 4 — Identifiers with special meaning

| audit | final | module |
| axiom | import | override |

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

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

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

5.11 Keywords

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` (9.11.1):

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

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

5.12 Operators and punctuators

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

```
preprocessing-op-or-punc: one of
{ } [ ] # ## ( ) <; :> <% %> %: %%; ; ; * -> -* . . new delete ? :: . .* -> ->* ~ ! + - * / % ^ & | = += -= *= /= %= ^= &= |= == != < > <= >= <=> && || <<= >>= ++ -- , and or xor not bitand bitor compl and_eq or_eq xor_eq not_eq
```

Each `preprocessing-op-or-punc` is converted to a single token in translation phase 7 (5.2).

§ 5.12
Table 5 — Keywords

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

Table 6 — Alternative representations

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

§ 5.13 Literals

5.13.1 Kinds of literals

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

literal:
- integer-literal
- character-literal
- floating-literal
- string-literal
- boolean-literal
- pointer-literal
- user-defined-literal

5.13.2 Integer literals

integer-literal:
- binary-literal integer-suffix\(^{\text{opt}}\)
- octal-literal integer-suffix\(^{\text{opt}}\)
- decimal-literal integer-suffix\(^{\text{opt}}\)
- hexadecimal-literal integer-suffix\(^{\text{opt}}\)

binary-literal:
- 0b binary-digit
- 0B binary-digit
- binary-literal '\(^{\text{opt}}\) binary-digit

octal-literal:
- 0 octal-digit\(^{\text{opt}}\)

decimal-literal:
- nonzero-digit
- decimal-literal '\(^{\text{opt}}\) digit

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

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

An integer literal is a sequence of digits that has no period or exponent part, with optional separating single quotes that are ignored when determining its value. An integer literal may have a prefix that specifies its base and a suffix that specifies its type. The lexically first digit of the sequence of digits is the most significant. A binary integer literal (base two) begins with 0b or 0B and consists of a sequence of binary digits. An octal integer literal (base eight) begins with the digit 0 and consists of a sequence of octal digits. A decimal integer literal (base ten) begins with a digit other than 0 and consists of a sequence of decimal digits. A hexadecimal integer literal (base sixteen) begins with 0x or 0X and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters a through f and A through F with decimal values ten through fifteen. [Example: The number twelve can be written 12, 014, 0XC, or 0b1100. The integer literals 1048576, 1’048’576, 0X100000, 0x10’0000, and 0’004’000’000 all have the same value. — end example]

The type of an integer literal is the first of the corresponding list in Table 7 in which its value can be represented.

Table 7 — Types of integer literals

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal literal</th>
<th>Binary, octal, or hexadecimal literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>long int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>u or U</td>
<td>unsigned int</td>
<td>unsigned int</td>
</tr>
<tr>
<td></td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>l or L</td>
<td>long int</td>
<td>long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>long long int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

17) The digits 8 and 9 are not octal digits.
Table 7 — Types of integer literals (continued)

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Decimal literal</th>
<th>Binary, octal, or hexadecimal literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both u or U and l or L</td>
<td>unsigned long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>1l or ll</td>
<td>long long int</td>
<td>long long int</td>
</tr>
<tr>
<td>Both u or U and ll or LL</td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

3 If an integer literal cannot be represented by any type in its list and an extended integer type (6.7.1) 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

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

encoding-prefix: one of
  u8  u  U  L

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

c-char:
  any member of the 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 \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 is one or more characters enclosed in single quotes, as in 'x', optionally preceded by u8, u, U, or L, as in u8'w', u'x', U'y', or L'z', respectively.

2 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.

3 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 0x0-0x7F (inclusive). — 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.

4 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 0x0-0xFFFF (inclusive). — 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.

5 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.

6 A character literal that begins with the letter L, such as L'z', is a wide-character literal. A wide-character literal has type wchar_t. The value of a wide-character literal containing a single c-char has value equal to the numerical value of the encoding of the c-char in the execution wide-character set, unless the c-char has no representation in the execution wide-character set, in which case the value is implementation-defined. [Note: The type wchar_t is able to represent all members of the execution wide-character set (see 6.7.1). — end note] The value of a wide-character literal containing multiple c-chars is implementation-defined.

7 Certain non-graphic characters, the single quote ', the double quote "", the question mark ?, and the backslash \, can be represented according to Table 8. 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 8 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

Table 8 — Escape sequences

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>new-line</td>
<td>NL(LF) \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</td>
<td>BEL \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 \xhhh</td>
</tr>
</tbody>
</table>

8 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 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]

9 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

---

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

floating-literal:  
decimal-floating-literal  
hexadecimal-floating-literal  

decimal-floating-literal:  
fractional-constant exponent-part

sign opt digit-sequence exponent-part floating-suffix opt  
digit-sequence exponent-part floating-suffix opt  

hexadecimal-floating-literal:  
hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffix opt  
hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-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-suffix: one of  
f 1 F L  

1 A floating literal consists of an optional prefix specifying a base, an integer part, a radix point, a fraction part, an \( e \), \( E \), \( p \) or \( P \), an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits if there is no prefix, or hexadecimal (base sixteen) digits if the prefix is 0x or 0X. The floating literal is a decimal floating literal in the former case and a hexadecimal floating literal in the latter case. Optional separating single quotes in a digit-sequence or hexadecimal-digit-sequence are ignored when determining its value. [Example: The floating literals 1.602'176'565e-19 and 1.602176565e-19 have the same value. — end example] Either the integer part or the fraction part (not both) can be omitted. Either the radix point or the letter \( e \) or \( E \) and the exponent (not both) can be omitted from a decimal floating literal. The radix point (but not the exponent) can be omitted from a hexadecimal floating literal. The integer part, the optional radix point, and the optional fraction part, form the significand of the floating literal. In a decimal floating literal, the exponent, if present, indicates the power of 10 by which the significand is to be scaled. In a hexadecimal floating literal, the exponent indicates the power of 2 by which the significand is to be scaled. [Example: The floating literals 49.625 and 0xC.68p+2 have the same value. — end example] If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is double unless explicitly specified by a suffix. The suffixes f and F specify float, the suffixes l and L specify long double. If the scaled value is not in the range of representable values for its type, the program is ill-formed.  

5.13.5 String literals  

string-literal:  
encoding-prefix opt " s-char-sequence opt "  
encoding-prefix opt R raw-string  

§ 5.13.5
s-char-sequence:
  s-char
  s-char-sequence s-char

s-char:
  any member of the 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:
  r-char
  r-char-sequence r-char

r-char:
  any member of the source character set, except a right parenthesis ) followed by
  the initial d-char-sequence (which may be empty) followed by a double quote ".

d-char-sequence:
  d-char
  d-char-sequence d-char

d-char:
  any member of the basic source character set except:
    space, the left parenthesis (, the right parenthesis ), the backslash \, and the control characters
    representing horizontal tab, vertical tab, form feed, and newline.

1 A string-literal is a sequence of characters (as defined in 5.13.3) surrounded by double quotes, optionally
  prefixed by R, u8, u8R, u, uR, U, UR, L, or LR, as in "...", R"(...)", u8"...", u8R"**(...)**", u"...",
  uR"**(...)**", u"...", UR"zzz(...)zzz", L"...", or LR"(...)", respectively.

2 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.

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

4 [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\n b
c)";
    assert(std::strcmp(p, "a\\n\nb\nc") == 0);

  — end note]

5 [Example: The raw string

    R"a(
    )\n    a"

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

    R"(x = \"y\""

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

6 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 (6.6.5), and is initialized with the given characters.

7 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 (6.7) has the value of the corresponding code unit of the UTF-8 encoding of the
  string.

8 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals.
A string-literal that begins with u, such as u"asdf", is a UTF-16 string literal. A UTF-16 string literal has type “array of n const char16_t”, where n is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-16 encoding of the string. [Note: A single c-char may produce more than one char16_t character in the form of surrogate pairs. A surrogate pair is a representation for a single code point as a sequence of two 16-bit code units. — end note]

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

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

In translation phase 6 (5.2), adjacent string-literals are concatenated. If both string-literals have the same encoding-prefix, the resulting concatenated string literal has that encoding-prefix. If one string-literal has no encoding-prefix, it is treated as a string-literal of the same encoding-prefix as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally-supported with implementation-defined behavior. [Note: 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 9 has some examples of valid concatenations.

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

Characters in concatenated strings are kept distinct.

[Example:

```
\"x\xA\" \"B\"
```

contains the two characters \x\xA and \x\xB after concatenation (and not the single hexadecimal character \x\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 ‘\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 ‘\0’. [Note: The size of a char16_t string literal is the number of code units, not the number of characters. — end note] Within char32_t and char16_t string literals, any universal-character-names shall be within the range 0x0 to 0x10FFFFF. 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

`bool literal`: `false` `true`

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

5.13.7 Pointer literals

`pointer literal`: `nullptr`

1 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-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-literal`:
- `fractional-constant exponent-part 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`

1 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.

2 A `user-defined-literal` is treated as a call to a literal operator or literal operator template (12.5.8). 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 (6.4.1). Let `S` be the set of declarations found by this lookup. `S` shall not be empty.

3 If `L` is a `user-defined-integer-literal`, let `n` be the literal without its `ud-suffix`. If `S` contains a literal operator with parameter type `unsigned long long`, the literal `L` is treated as a call of the form

```cpp
operator "" X(nULL)
```

Otherwise, `S` shall contain a raw literal operator or a numeric literal operator template (12.5.8) but not both. If `S` contains a raw literal operator, the literal `L` is treated as a call of the form

```cpp
operator "" X("n")
```

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

```cpp
operator "" X<"c1", "c2", ... "ck">()
```

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 \textit{user-defined-floating-literal}, let \( f \) be the literal without its \textit{ud-suffix}. If \( S \) contains a literal operator with parameter type \texttt{long double}, the literal \( L \) is treated as a call of the form
\[
\text{operator "} X(fL) \text{"}
\]
Otherwise, \( S \) shall contain a raw literal operator or a numeric literal operator template (12.5.8) but not both. If \( S \) contains a raw literal operator, the literal \( L \) is treated as a call of the form
\[
\text{operator "} X("f") \text{"}
\]
Otherwise (\( S \) contains a numeric literal operator template), \( L \) is treated as a call of the form
\[
\text{operator "} X<\text{c}_1, \text{c}_2, \ldots, \text{c}_k>()(f)
\]
where \( f \) is the source character sequence \( \text{c}_1\text{c}_2\ldots\text{c}_k \). \[\text{Note: The sequence } \text{c}_1\text{c}_2\ldots\text{c}_k \text{ can only contain characters from the basic source character set.} \quad \text{—end note}\]

If \( L \) is a \textit{user-defined-string-literal}, let \( \text{str} \) be the literal without its \textit{ud-suffix} and let \( \text{len} \) be the number of code units in \( \text{str} \) (i.e., its length excluding the terminating null character). If \( S \) contains a literal operator template with a non-type template parameter for which \( \text{str} \) is a well-formed \textit{template-argument}, the literal \( L \) is treated as a call of the form
\[
\text{operator "} X<\text{str}>()(\text{str}, \text{len}) \text{"}
\]
Otherwise, the literal \( L \) is treated as a call of the form
\[
\text{operator "} X<\text{str}>()(\text{str}) \text{"}
\]
If \( L \) is a \textit{user-defined-character-literal}, let \( \text{ch} \) be the literal without its \textit{ud-suffix}. \( S \) shall contain a literal operator (12.5.8) whose only parameter has the type of \( \text{ch} \) and the literal \( L \) is treated as a call of the form
\[
\text{operator "} X(\text{ch}) \text{"}
\]

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

In translation phase 6 (5.2), adjacent string literals are concatenated and \textit{user-defined-string-literals} are considered string literals for that purpose. During concatenation, \textit{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 \textit{user-defined-string-literal}, all the participating \textit{user-defined-string-literals} shall have the same \textit{ud-suffix} and that suffix is applied to the result of the concatenation.

\[\text{Example:}\]
\[
\text{int main}() \{ \\
\quad \text{L"A" } \text{"B" } \text{"C"}_x; \quad \text{// OK: same as L"ABC"}_x \\
\quad \text{"P"}_x \text{"Q" } \text{"R"}_y; \quad \text{// error: two different ud-suffixes}
\}
\]
\[\text{—end example}\]
6 Basics

[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 (12.5), literal-operator-id (12.5.8), conversion-function-id (11.3.7.2), or template-id (13.2) that denotes an entity or label (8.6.4, 8.1).

Every name that denotes an entity is introduced by a declaration. Every name that denotes a label is introduced either by a goto statement (8.6.4) or a labeled-statement (8.1).

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 (6.6.5.3), a structured binding (9.5) 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 (6.4).

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 (13.5), or
— they are the names of literal operators (12.5.8) 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 (6.5) of the name specified in each translation unit.

6.1 Declarations and definitions

A declaration (Clause 9) may introduce one or more names into a translation unit or redeclare 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 (Clause 9),
— controlling template instantiation (13.8.2),
— guiding template argument deduction for constructors (13.10),
— use of attributes (Clause 9), 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 (9.4),
— it contains the extern specifier (9.1.1) or a linkage-specification (9.10) and neither an initializer nor a function-body,

Appearing inside the brace-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.
it declares a non-inline static data member in a class definition (11.3, 11.3.8),

it declares a static data member outside a class definition and the variable was defined within the class with the `constexpr` specifier (this usage is deprecated; see D.4),

it is introduced by an `elaborated-type-specifier` (11.2),

it is an `opaque-enum-declaration` (9.6),

it is a `template-parameter` (13.1),

it is a `parameter-declaration` (9.2.3.5) in a function declarator that is not the declarator of a function-definition,

it is a `typedef` declaration (9.1.3),

it is an `alias-declaration` (9.1.3),

it is a `using-declaration` (9.8),

it is a `deduction-guide` (13.10),

it is a `static_assert-declaration` (Clause 9),

it is an `attribute-declaration` (Clause 9),

it is an `empty-declaration` (Clause 9),

it is an explicit instantiation declaration (13.8.2), or

it is an explicit specialization (13.8.3) whose `declaration` is not a definition.

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

```c
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 X
    static int y; // declares static data member y
    X(): x(0) { } // defines a constructor of X
};
int X::y = 1; // defines X::y
enum { up, down }; // defines up and down
namespace N { int d; } // defines N and N::d
namespace N1 = N; // defines N1
X anX; // defines anX
```

whereas these are just declarations:

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

---

3 [Note: In some circumstances, C++ implementations implicitly define the default constructor (11.3.4.1), copy constructor, move constructor (11.3.4.2), copy assignment operator, move assignment operator (11.3.5), or destructor (11.3.6) member functions. — end note] [Example: Given

```c
#include <string>

struct C {
    std::string s; // std::string is the standard library class (Clause 21)
};
```
int main() {
    C a;
    C b = a;
    b = a;
}

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

struct C {
    std::string s;
    C() : s() { }
    C(const C& x): s(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

4 [Note: A class name can also be implicitly declared by an elaborated-type-specifier (9.1.7.3). — end note]
as part of forming that expression or conversion, unless it is a pure virtual function and either the expression is not an \textit{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 (11.3.7.2), allocation functions for \textit{new-expressions} (7.6.2.5), as well as non-default initialization (9.3). 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 (11.9.5). — end note]

(3.2) — A deallocation function for a class is named by a \textit{new-expression} if it is the single matching deallocation function for the allocation function selected by overload resolution, as specified in 7.6.2.5.

(3.3) — A deallocation function for a class is named by a \textit{delete-expression} if it is the selected usual deallocation function as specified in 7.6.2.6 and 11.11.

4 A variable \(x\) whose name appears as a potentially-evaluated expression \(e\) is \textit{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 \texttt{*this} is odr-used if \texttt{this} appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (11.3.2)).

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 (11.3.6).

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 11.3.5. A constructor for a class is odr-used as specified in 9.3. A destructor for a class is odr-used if it is potentially invoked (11.3.6).

9 A local entity (Clause 6) is \textit{odr-usable} in a declarative region (6.3.1) if:

(9.1) — either the local entity is not \texttt{*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 (6.3.1) between the point at which the entity is introduced and the region (where \texttt{*this} is considered to be introduced within the innermost enclosing class or non-lambda function definition scope), either:

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

(9.2.2) — the intervening declarative region is the function parameter scope of a \textit{lambda-expression} that has a \textit{simple-capture} naming the entity or has a \textit{capture-default}, and the block scope of the \textit{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:

\begin{verbatim}
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); // error: n is not odr-usable due to intervening function parameter scope
    [=](int k = n) {};
    // error: n is not odr-usable due to being
\end{verbatim}

21) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.
Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement (8.4.1); 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 11.3.4.1, 11.3.4.2, 11.3.6, and 11.3.5). A definition of an inline function or variable shall be reachable in every translation unit in which it is odr-used outside of a discarded statement. [Example:

```cpp
class A {}
A a = A();
```  
A program containing this translation unit is ill-formed because `A` is odr-used but not defined, and cannot be defined in any other translation unit because the local class `A` cannot be named outside this translation unit. —end example]

A definition 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`:

```cpp
struct X;
X x1;
X x2;
```  
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]

There can be more than one definition of a class type (Clause 11), enumeration type (9.6), inline function with external linkage (9.1.6), inline variable with external linkage (9.1.6), class template (Clause 13), non-static function template (13.6.6), concept (13.6.8), static data member of a class template (13.6.1.3), member function of a class template (13.6.1.1), or template specialization for which some template parameters are not specified (13.8, 13.6.5) in a program provided that no prior definition is necessarily reachable (10.6) at the point where a definition appears, and provided the definitions satisfy the following requirements. There shall not be more than one definition of an entity that is attached to a named module (10.1); no diagnostic is required unless a prior definition is reachable at a point where a later definition appears. Given such an entity named `D` defined in more than one translation unit, then
(12.1) — each definition of D shall consist of the same sequence of tokens; and

(12.2) — in each definition of D, corresponding names, looked up according to 6.4, shall refer to an entity defined within the definition of D, or shall refer to the same entity, after overload resolution (12.3) and after matching of partial template specialization (13.9.3), except that a name can refer to

(12.2.1) — a non-volatile const object with internal or no linkage if the object

(12.2.1.1) — has the same literal type in all definitions of D,

(12.2.1.2) — is initialized with a constant expression (7.7),

(12.2.1.3) — is not odr-used in any definition of D, and

(12.2.1.4) — has the same value in all definitions of D,

or

(12.2.2) — 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;

and

(12.3) — in each definition of D, corresponding entities shall have the same language linkage; and

(12.4) — in each definition of D, 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, or to a function defined within the definition of D; and

(12.5) — in each definition of D, a default argument used by an (implicit or explicit) function call is treated as if its token sequence were present in the definition of D; that is, the default argument is subject to the requirements described in this paragraph (and, if the default argument has subexpressions with default arguments, this requirement applies recursively)\(^\text{22}\); and

(12.6) — if D invokes a function with a precondition, or is a function that contains an assertion or has a contract condition (9.11.4), it is implementation-defined under which conditions all definitions of D shall be translated using the same build level and violation continuation mode; and

(12.7) — if D is a class with an implicitly-declared constructor (11.3.4.1, 11.3.4.2), 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;
};
D d1;               //X(int, int) called by D()

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

— end example]
```

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 (13.7.3), and also to dependent names at the point of instantiation (13.7.2). If the definitions of D satisfy all these requirements,

\(^{22}\) 9.2.3.6 describes how default argument names are looked up.

§ 6.2
then the behavior is as if there were a single definition of \( D \). [Note: The entity is still declared in multiple translation units, and 6.5 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. — end note] If the definitions of \( D \) do not satisfy these requirements, then the behavior is undefined.

### 6.3 Scope

#### 6.3.1 Declarative regions and scopes

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 contiguous 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 (11.8.3), certain uses of the elaborated-type-specifier (9.1.7.3), and using-directives (9.7.3) alter this general behavior.

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

- they shall all refer to the same entity, or all refer to functions and function templates; or
- 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 (6.3.10). [Note: A structured binding (9.5), namespace name (9.7), or class template name (Clause 13) 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 (9.1.7.3) and friend declarations (11.8.3) may introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (6.5) 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 6.4. — end note]

#### 6.3.2 Point of declaration

The point of declaration for a name is immediately after its complete declarator (9.2) and before its initializer (if any), except as noted below. [Example:

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

Here the second \( x \) is initialized with its own (indeterminate) value. — 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]

3 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* (Clause 11). The point of declaration for an enumeration is immediately after the *identifier* (if any) in either its *enum-specifier* (9.6) or its first *opaque-enum-declaration* (9.6), 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* (9.8).

5 The point of declaration for an enumerator is immediately after its *enumerator-definition*. [Example:

```c
const int x = 12;
{ enum { x = x }; }
```
Here, the enumerator x is initialized with the value of the constant x, namely 12. —end example]

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,

```c
struct X {
    enum E { z = 16 };
    int b[X::z];        // OK
};
```
—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 6.4.4 and 9.1.7.3. —end note]

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

(7.1) — for a declaration of the form

```
class-key attribute-specifier-seq opt identifier ;
```
the *identifier* is declared to be a *class-name* in the scope that contains the declaration, otherwise

(7.2) — for an *elaborated-type-specifier* of the form

```
class-key identifier
```
if the *elaborated-type-specifier* is used in the *decl-specifier-seq* or *parameter-declaration-clause* of a function defined in namespace scope, the *identifier* is declared as a *class-name* in the namespace that contains the declaration; otherwise, except as a friend declaration, the *identifier* is declared in the smallest namespace or block scope that contains the declaration. [Note: 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 6.4.4 and 9.1.7.3. —end note]

8 The point of declaration for an injected-class-name (Clause 11) is immediately following the opening brace of the class definition.

9 The point of declaration for a function-local predefined variable (9.4.1) is immediately before the *function-body* of a function definition.

10 The point of declaration of a structured binding (9.5) 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 (8.5.4) is immediately after the *for-range-initializer*.

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

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

§ 6.3.2
[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 (9.7.1.2). 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]

### 6.3.3 Block scope

A name declared in a block (8.3) is local to that block; it has **block scope**. Its potential scope begins at its point of declaration (6.3.2) and ends at the end of its block. A variable declared at block scope is a **local variable**.

1. The name declared in an `exception-declaration` is local to the `handler` and shall not be redeclared in the outermost block of the `handler`.
2. 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; see 8.4.

### 6.3.4 Function parameter scope

A function parameter (including one appearing in a `lambda-declarator`) or function-local predefined variable (9.4) 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`.

### 6.3.5 Function scope

Labels (8.1) have **function scope** and may be used anywhere in the function in which they are declared. Only labels have function scope.

### 6.3.6 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 (6.3.2) onwards; and for each `using-directive` (9.7.3) 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:

```cpp
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 l+a; // l is from unnamed namespace
    }

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

§ 6.3.6 32
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:

```cpp
export module Q;
export int sq(int i) { return i*i; }
```

Translation unit #2:

```cpp
export module R;
export import Q;
```

Translation unit #3:

```cpp
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 6.4.3.2.

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 (6.3.2) 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.

### 6.3.7 Class scope

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 (11.3) 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 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 (9.2.3.6)).

5 [Example:

```cpp
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
}```

§ 6.3.7 33
The name of a class member shall only be used as follows:

- in the scope of its class (as described above) or a class derived (11.6) from its class,
- after the . operator applied to an expression of the type of its class (7.6.1.4) or a class derived from its class,
- after the -> operator applied to a pointer to an object of its class (7.6.1.4) or a class derived from its class,
- after the :: scope resolution operator (7.5.4.2) applied to the name of its class or a class derived from its class.

6.3.8 Enumeration scope

The name of a scoped enumerator (9.6) has enumeration scope. Its potential scope begins at its point of declaration and terminates at the end of the enum-specifier.

6.3.9 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:

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 (6.3.2) 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,

    template<class T, T* p, class U = T> class X { /* ... */
    template<class T> void f(T* p = new T);

    This also implies that a template-parameter can be used in the specification of base classes. For example,

    template<class T> class X : public Array<T> { /* ... */
}
template<class T> class Y : public T { /* ... */ };  

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

4 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 (6.3.10).] [Example:  

```cpp
typedef int N;
template<N X, typename N, template<N Y> class T> struct A;  
```

Here, X is a non-type template parameter of type int and Y is a non-type template parameter of the same type as the second template parameter of A. —end example] —end note]

5 [Note: Because the name of a template parameter cannot be redeclared within its potential scope (13.7.1), a template parameter’s scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 13.7.1. —end note]

### 6.3.10 Name hiding

A declaration of a name in a nested declarative region hides a declaration of the same name in an enclosing declarative region; see 6.3.1 and 6.4.1.

2 If a class name (11.2) or enumeration name (9.6) 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 (6.4.1)), 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 6.3.7. The declaration of a member in a derived class (11.6) hides the declaration of a member of a base class of the same name; see 11.7.

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 6.4.3.2.

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

### 6.4 Name lookup

The name lookup rules apply uniformly to all names (including typedef-names (9.1.3), namespace-names (9.7), and class-names (11.2)) 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 (6.1) of that name. The declarations found by name lookup shall either all denote the same entity or shall all denote functions or function templates; in the latter case, the declarations are said to form a set of overloaded functions (12.1). Overload resolution (12.3) takes place after name lookup has succeeded. The access rules (11.8) 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 (10.6) 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 (Clause 11) is also considered to be a member of that class for the purposes of name hiding and lookup.

4 [Note: 6.5 discusses linkage issues. The notions of scope, point of declaration and name hiding are discussed in 6.3. —end note]

### 6.4.1 Unqualified name lookup

In all the cases listed in 6.4.1, 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 9.7.3. For the purpose of the unqualified name lookup rules described in 6.4.1, the declarations from the namespace nominated by the using-directive are considered members of that enclosing namespace.

§ 6.4.1 35
The lookup for an unqualified name used as the postfix-expression of a function call is described in 6.4.2. [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 13.2). 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 6.4.2 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 (6.4.2) 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 (8.3) 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 (11.3) 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 (11.7), or

23) 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.

24) 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 \( Y \) (11.3.10), 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),\(^{25}\) or

— if \( X \) is a local class (11.5) 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 9.7.1.2. — end note]  [Note: 6.3.7 further describes the restrictions on the use of names in a class definition. 11.3.10 further describes the restrictions on the use of names in nested class definitions. 11.5 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 (11.3) of \( X \) or in the definition of a class member outside of the definition of \( X \), following the member's declarator-id\(^{26}\), 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 (8.3), or

— shall be a member of class \( X \) or be a member of a base class of \( X \) (11.7), or

— if \( X \) is a nested class of class \( Y \) (11.3.10), 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).\(^{27}\) or

— if \( X \) is a local class (11.5) 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();
        };
    }
}
```

25) This lookup applies whether the definition of \( X \) is nested within \( Y \)'s definition or whether \( X \)'s definition appears in a namespace scope enclosing \( Y \)'s definition (11.3.10).

26) That is, an unqualified name that occurs, for instance, in a type in the parameter-declaration-clause or in the noexcept-specifier.

27) This lookup applies whether the member function is defined within the definition of class \( X \) or whether the member function is defined in a namespace scope enclosing \( X \)'s definition.
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: 11.3.1 and 11.3.8 further describe the restrictions on the use of names in member function definitions. 11.3.10 further describes the restrictions on the use of names in the scope of nested classes. 11.5 further describes the restrictions on the use of names in local class definitions. — end note]

9 Name lookup for a name used in the definition of a friend function (11.8.3) 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.

10 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 is first looked up in the scope of the member function’s class (11.7). 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:

```c
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 (9.2.3.6) in a function parameter-declaration-clause or used in the expression of a mem-initializer for a constructor (11.9.2), 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: 9.2.3.6 further describes the restrictions on the use of names in default arguments. 11.9.2 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 (11.3.8.2) (after the qualified-id of the static member) is looked up as if the name was used in a member function of X. [Note: 11.3.8.2 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:

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

—end example]

A name used in the handler for a function-try-block (Clause 14) 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]

[Note: The rules for name lookup in template definitions are described in 13.7. — end note]

### 6.4.2 Argument-dependent name lookup

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 (6.4.1) may be searched, and in those namespaces, namespace-scoped friend function or function template declarations (11.8.3) 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]
```

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 (9.7.1), 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 a set of overloaded functions and/or function templates, 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 set of overloaded functions 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 (6.4.1) 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 {
    public:
        void f(T);
        void g(T, int);
    }

    void g(NS::T, float);
}

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

— end example]

When considering an associated namespace \( N \), the lookup is the same as the lookup performed when \( N \) is used as a qualifier (6.4.3.2) 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 (11.8.3) 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 (9.7.1.2).
4. Any declaration \( D \) in \( N \) that is in the interface 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 (13.7.2, 13.7.4.2), any declaration \( D \) in \( N \) is visible if \( D \) would be visible to qualified name lookup (6.4.3.2) at any point in the instantiation context (10.5) 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(X, X);
}
```

Translation unit #2:
```cpp
export module N;
```
import M;
export R::X make();
namespace R { static int g(X); }
template<
typename T,

typename U>

void apply(T t, U u) {
    f(t, u);
    g(t);
}

Translation unit #3:

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); // ill-formed: R and R::f are not visible here
    f(x); // OK, calls R::f from interface of M
    f(x, S::Z()); // ill-formed: S::f in module M not considered
    // even though S is an associated namespace
    apply(x, S::Z()); // OK, S::f is visible in instantiation context, and
    // R::g is visible even though it has internal linkage
}

— end example]
is visible in global scope because of a using-directive (6.4.3.2). The use of :: allows a global name to be referred to even if its identifier has been hidden (6.3.10).

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:

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

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

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

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

— end example] [Note: 6.4.5 describes how name lookup proceeds after the , and -> operators. — end note]

6.4.3.1 Class members [class.qual]

If the nested-name-specifier of a qualified-id nominates a class, the name specified after the nested-name-specifier is looked up in the scope of the class (11.7), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (11.6). [Note: A class member can be referred to using a qualified-id at any point in its potential scope (6.3.7). — end note] The exceptions to the name lookup rule above are the following:

1. (1.1) the lookup for a destructor is as specified in 6.4.3;

2. (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 6.4.5);

3. (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.

4. (1.4) the lookup for a name specified in a using-declaration (9.8) also finds class or enumeration names hidden within the same scope (6.3.10).

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

1. (2.1) if the name specified after the nested-name-specifier, when looked up in C, is the injected-class-name of C (Clause 11), or

2. (2.2) in a using-declarator of a using-declaration (9.8) 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:

\begin{verbatim}
struct A { A(); };  
struct B: public A { B(); }; 
\end{verbatim}

28) 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]

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

6.4.3.2 Namespace members

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

2 For a namespace X and name m, the namespace-qualified lookup set S(X,m) is defined as follows: Let S'(X,m) be the set of all declarations of m in X and the inline namespace set of X (9.7.1) whose potential scope (6.3.6) would include the namespace in which m is declared at the location of the nested-name-specifier. If S'(X,m) is not empty, S(X,m) is S'(X,m); otherwise, S(X,m) is the union of S(N_i,m) for all namespaces N_i nominated by using-directives in X and its inline namespace set.

3 Given X::m (where X is a user-declared namespace), or given ::m (where X is the global namespace), if S(X,m) is the empty set, the program is ill-formed. Otherwise, if S(X,m) has exactly one member, or if the context of the reference is a using-declaration (9.8), S(X,m) is the required set of declarations of m. Otherwise if the use of m is not one that allows a unique declaration to be chosen from S(X,m), the program is ill-formed. [Example:

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

namespace Z {
  void h(double);
}

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

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

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

void h() {
  AB::g();          // g is declared directly in AB, therefore S is {AB::g()} and AB::g() is chosen
  AB::f(1);          // f is not declared directly in AB so the rules are applied recursively to A and B;
}
namespace Y is not searched and Y::f(float) is not considered;
// S is {A::f(int), B::f(char)} and overload resolution chooses A::f(int)

AB::f('c'); // as above but resolution chooses B::f(char)

AB::x++; // x is not declared directly in AB, and is not declared in A or B, so the rules
// are applied recursively to Y and Z, S is {} so the program is ill-formed

AB::i++; // i is not declared directly in AB so the rules are applied recursively to A and B,
// S is {A::i, B::i} so the use is ambiguous and the program is ill-formed

AB::h(16.8); // h is not declared directly in AB and not declared directly in A or B so the rules
// are applied recursively to Y and Z, S is {Y::h(int), Z::h(double)} and
// overload resolution chooses Z::h(double)

— end example

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

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]

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]

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 (9.7.1) of that namespace. [Example:

namespace A {
  namespace B {
    void f1(int);
  }
  using namespace B;
}
void A::f1(int);  // ill-formed, 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);
  }
  using namespace B;
}

— end example]
namespace C {
    namespace D {
        void f1(int);
    }
}

using namespace A;
using namespace C::D;
void B::f1(int){ } // OK, defines A::B::f1(int)

—end example

6.4.4 Elaborated type specifiers

An elaborated-type-specifier (9.1.7.3) 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 (6.3.10).

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-seqopt identifier;
```

the identifier is looked up according to 6.4.1 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-seqopt identifier;
```

the elaborated-type-specifier is a declaration that introduces the class-name as described in 6.3.2.

If the elaborated-type-specifier has a nested-name-specifier, qualified name lookup is performed, as described in 6.4.3, 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:

```
struct Node {
    struct Node* Next;
    struct Data* Data; // OK: Refers to Node at global scope
};

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 (9.1.7.3)
    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 (9.1.7.3)
struct Base::Data; // error: cannot introduce a qualified type (9.1.7.3)
struct Base::Datum; // error: Datum undefined
struct Base::Data* pBase; // OK: refers to nested Data

—end example

6.4.5 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 (13.2) or a less-than operator. The identifier is first looked up in the class of the object expression (11.7). If the identifier is not found, it is then looked up in the context of the entire postfix-expression and shall name a class template.

2 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 (11.7).

3 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:

```c
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]

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

```c
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 (11.7) and the name, if found, is used. Otherwise it is looked up in the context of the entire postfix-expression. [Note: See 6.4.3, which describes the lookup of a name before ::, which will only find a type or namespace name. — end note]

5 If the qualified-id has the form

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

6 If the nested-name-specifier contains a simple-template-id (13.2), the names in its template-arguments are looked up in the context in which the entire postfix-expression occurs.

7 If the id-expression is a conversion-function-id, its conversion-type-id is first looked up in the class of the object expression (11.7) 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:

```c
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]

### 6.4.6 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.

### 6.5 Program and linkage

A program consists of one or more translation units (Clause 5) linked together. A translation unit consists of a sequence of declarations.
A private-module-fragment shall appear only in a primary module interface unit (10.1). A module unit with a private-module-fragment shall be the only module unit of its module; no diagnostic is required.

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 (6.3.6) has internal linkage if it is the name of

- a variable, variable template, function, or function template that is explicitly declared static; or
- a non-template variable of non-volatile const-qualified type, unless
  - it is explicitly declared extern, or
  - it is inline or exported, or
  - it was previously declared and the prior declaration did not have internal linkage; or
- 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

- a variable; or
- a function; or
- a named class (Clause 11), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (9.1.3); or
- a named enumeration (9.6), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (9.1.3); or
- a template
  has its linkage determined as follows:
  - if the enclosing namespace has internal linkage, the name has internal linkage;
  - otherwise, if the declaration of the name is attached to a named module (10.1) and is not exported (10.2), the name has module linkage;
  - otherwise, the name has external linkage.
In addition, a member function, static data member, a named class or enumeration of class scope, or an
unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration
has the typedef name for linkage purposes (9.1.3), 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]

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
}

void q() { /* ... */ }  // some other, unrelated q
```
—end example]

Names not covered by these rules have no linkage. Moreover, except as noted, a name declared at block
scope (6.3.3) has no linkage.

Two names that are the same (Clause 6) and that are declared in different scopes shall denote the same
variable, function, type, template or namespace if

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

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

(11.3) — when both names denote functions, the parameter-type-lists of the functions (9.2.3.5) are identical; and

(11.4) — when both names denote function templates, the signatures (13.6.6.1) 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. — end note]

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

"decls.h":

```c
int f(); // #1, attached to the global module
int g(); // #2, attached to the global module
```

Module interface of M:

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

Other translation unit:

```c
import M;
static int h(); // error: matches #3
int k(); // error: matches #4
```

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

After all adjustments of types (during which typedefs (9.1.3) 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 (9.2.3.4). 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 (9.10). — end note]

### 6.6 Memory and objects

#### 6.6.1 Memory model

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

[Note: The representation of types is described in 6.7. — end note]

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 (6.8.2) can access separate memory locations without interfering with each other.

[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]

[Example: A class declared as

```c
struct {
    char a;
```]

The number of bits in a byte is reported by the macro `CHAR_BIT` in the header `<climits>`.

---

29) The number of bits in a byte is reported by the macro `CHAR_BIT` in the header `<climits>`.

§ 6.6.1 50
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]

6.6.2 Object model

The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object
is created by a definition (6.1), by a new-expression (7.6.2.5), when implicitly changing the active member of a
union (11.4), or when a temporary object is created (7.3.4, 6.6.7). An object occupies a region of storage
in its period of construction (11.9.4), throughout its lifetime (6.6.3), and in its period of destruction (11.9.4).
[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 (Clause 6). An object has a storage duration (6.6.5) which influences its lifetime (6.6.3). An object has
a type (6.7). Some objects are polymorphic (11.6.2); 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.

Objects can contain other objects, called subobjects. A subobject can be a member subobject (11.3), a base
class subobject (11.6), 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).

[Note: If the subobject contains a reference member or a const subobject, the name of the original subobject
cannot be used to access the new object (6.6.3). — end note] [Example:

    struct X { const int n; };
    union U { X x; float f; };
    void tong() {
      U u = {{ 1 }};  // OK, creates new subobject of u (11.4)
      u.f = 5.f;      // OK, creates new subobject of u
      X *p = new (&u.x) X {2};
      assert(p->n == 2); // OK
      assert(*std::launder(&u.x.n) == 2); // OK
      assert(u.x.n == 2); // undefined behavior, u.x does not name new subobject
    }

    — end example]

If a complete object is created (7.6.2.5) 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.

[Note: If that portion of the array previously provided storage for another object, the lifetime of that object
ends because its storage was reused (6.6.3). — end note] [Example:

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

§ 6.6.2
```cpp
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
}
```

```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:

1. \( a \) is a subobject of \( b \), or
2. \( b \) provides storage for \( a \), or
3. there exists an object \( c \) where \( a \) is nested within \( c \), and \( c \) is nested within \( b \).

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

1. If \( x \) is a complete object, then the complete object of \( x \) is itself.
2. 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 (11.3), 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:

1. a base class subobject, or
2. a non-static data member declared with the no_unique_address attribute (9.11.11).

8 An object has nonzero size if it

1. is not a potentially-overlapping subobject, or
2. is not of class type, or
3. is of a class type with virtual member functions or virtual base classes, or
4. 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 (11.3.9), 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 (6.7) 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.

30 [Example:

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

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

[Note: C++ provides a variety of fundamental types and several ways of composing new types from existing types (6.7). --- end note]
6.6.3 Object and reference lifetime

1 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) (9.3),

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 (9.3.1, 11.9.2), or as described in 11.4. 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 (6.6.2).

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.

3 [Note: 11.9.2 describes the lifetime of base and member subobjects. — end note]

4 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 11.9.2 and in 11.9.4. 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. 11.9.2 and 11.9.4 describe the behavior of objects during the construction and destruction phases. — end note]

5 A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, 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 (7.6.2.6) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

6 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 11.9.4. Otherwise, such a pointer refers to allocated storage (6.6.5.4.1), 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 value may only be used in limited ways, as described below. The program has undefined behavior if:

(6.1) — 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,
(6.2) — the pointer is used to access a non-static data member or call a non-static member function of the object, or
(6.3) — the pointer is implicitly converted (7.3.11) to a pointer to a virtual base class, or
(6.4) — 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
(6.5) — 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();
```

31) For example, before the construction of a global object that is initialized via a user-provided constructor (11.9.4).
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
}

— end example

7 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 11.9.4. Otherwise, such a glvalue refers to allocated storage (6.6.5.4.1), 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
(7.2) — the glvalue is used to call a non-static member function of the object, or
(7.3) — the glvalue is bound to a reference to a virtual base class (9.3.3), 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:

(8.1) — the storage for the new object exactly overlays the storage location which the original object occupied, and
(8.2) — the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and
(8.3) — the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and
(8.4) — neither the original object nor the new object is a potentially-overlapping subobject (6.6.2).

[Example:

struct C {
    int i;
    void f();
    const C& operator=( const C& );
};

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

— end example] [Note: If these conditions are not met, a pointer to the new object can be obtained from a
pointer that represents the address of its storage by calling std::launder (17.6). — end note]

9 If a program ends the lifetime of an object of type T with static (6.6.5.1), thread (6.6.5.2), or automatic (6.6.5.3)
storage duration and if T has a non-trivial destructor, the program must ensure that an object of the
original type occupies that 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:

class T { }
struct B { 
 ~B();
};

void h() {
 B b;
 new (&b) T;
} // undefined behavior at block exit

— end example]

10 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:

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

 const B b;

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

— end example]

11 In this subclause, “before” and “after” refer to the “happens before” relation (6.8.2). [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]

6.6.4 Indeterminate values [basic.indet]

1 When storage for an object with automatic or dynamic storage duration is obtained, the object has an
indeterminate value, and if no initialization is performed for the object, that object retains an indeterminate
value until that value is replaced (7.6.19). [Note: Objects with static or thread storage duration are
zero-initialized, see 6.8.3.2. — 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 (6.7.1) 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

32) 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.
— a discarded-value expression (7.2.3),
then the result of the operation is an indeterminate value.

— 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.

— 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.

— 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:

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

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

Static, thread, and automatic storage durations are associated with objects introduced by declarations (6.1) and implicitly created by the implementation (6.6.7). The dynamic storage duration is associated with objects created by a new-expression (7.6.2.5).

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 (6.7.2). 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.33

6.6.5.1 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 shall last for the duration of the program (6.8.3.2, 6.8.3.4).

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 11.9.5.

The keyword static can be used to declare a local variable with static storage duration. [Note: 8.7 describes the initialization of local static variables; 6.8.3.4 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.

33) Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.
All variables declared with the `thread_local` keyword have thread storage duration. The storage for these entities shall last 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. A variable with thread storage duration shall be initialized before its first odr-use (6.2) and, if constructed, shall be destroyed on thread exit.

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 8.7. —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 11.9.5.

Objects can be created dynamically during program execution (6.8.1), using `new-expressions` (7.6.2.5), and destroyed using `delete-expressions` (7.6.2.6). 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 including the header `<new>` 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 including the appropriate header. —end note] Allocation and/or deallocation functions may also be declared and defined for any class (11.11).

If the behavior of an allocation or deallocation function does not satisfy the semantic constraints specified in 6.6.5.4.1 and 6.6.5.4.2, the behavior is undefined.

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 (9.2.3.6). 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

§ 6.6.5.4.1  57
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.

2 An allocation function attempts to allocate the requested amount of storage. If it is successful, it returns the address 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. For an allocation function other than a reserved placement allocation function (17.6.2.3), the pointer returned is suitably aligned so that it can be converted to a pointer to any suitable complete object type (17.6.2.1) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). 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 (7.3.11) \( p_0 \) different from any previously returned value \( p_1 \), unless that value \( p_1 \) was subsequently passed to a replaceable deallocation function. Furthermore, for the library allocation functions in 17.6.2.1 and 17.6.2.2, \( p_0 \) represents the address of a block of storage disjoint from the storage for any other object accessible to the caller. The effect of indirection through a pointer returned from a request for zero size is undefined.\(^{34}\)

3 An allocation function that fails to allocate storage can invoke the currently installed new-handler function (17.6.3.3), if any. \( {\text{[Note: A program-supplied allocation function can obtain the address of the currently installed new\textunderscore handler using the std: : } \text{get\textunderscore new\textunderscore handler function (17.6.3.5). \text{— end note}]} \) An allocation function that has a non-throwing exception specification (14.4) 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 (14.1) of a type that would match a handler (14.3) of type std: : bad\textunderscore alloc (17.6.3.1).

4 A global allocation function is only called as the result of a new expression (7.6.2.5), or called directly using the function call syntax (7.6.1.2), or called indirectly to allocate storage for a coroutine state (9.4.4), or called indirectly through calls to the functions in the C++ standard library. \( {\text{[Note: In particular, a global allocation function is not called to allocate storage for objects with static storage duration (6.6.5.1), for objects or references with thread storage duration (6.6.5.2), for objects of type std: : type\textunderscore info (7.6.1.7), or for an exception object (14.1). \text{— end note}]} \)

6.6.5.4.2 Deallocation functions \( {\text{[basic.stc.dynamic.deallocation]}} \)

1 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.

2 A deallocation function is a destroying operator delete if it has at least two parameters and its second parameter is of type std: : destroying\textunderscore delete\textunderscore t. A destroying operator delete shall be a class member function named operator delete. \( {\text{[Note: Array deletion cannot use a destroying operator delete. \text{— end note}]} \)

3 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

\[
\begin{align*}
(3.1) & \quad \text{optionally, a parameter of type std: : destroying\textunderscore delete\textunderscore t, then} \\
(3.2) & \quad \text{optionally, a parameter of type std: :size\textunderscore t, then} \\
(3.3) & \quad \text{optionally, a parameter of type std: :align\textunderscore val\textunderscore t.}
\end{align*}
\]

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.

4 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.

\(^{34}\) 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.

\(^{35}\) The global operator delete(void*, std: :size\textunderscore t) precludes use of an allocation function void operator new(std: :size\textunderscore t, std: :size\textunderscore t) as a placement allocation function (C.3.2).

\(\S~6.6.5.4.2\)
If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (7.3.11), the deallocation function shall deallocate the storage referenced by the pointer, ending the duration of the region of storage.

6.6.5.4.3 Safely-derived pointers

A traceable pointer object is

(1.1) an object of an object pointer type (6.7.2), or
(1.2) an object of an integral type that is at least as large as \texttt{std::intptr\_t}, or
(1.3) a sequence of elements in an array of narrow character type (6.7.1), where the size and alignment of the sequence match those of some object pointer type.

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

(2.1) the value returned by a call to the C++ standard library implementation of \texttt{::operator new(std::size\_t)} or \texttt{::operator new(std::size\_t, std::align\_val\_t)};\footnote{This subclause does not impose restrictions on indirection through pointers to memory not allocated by \texttt{::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 \texttt{std::malloc} is not restricted.}
(2.2) 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;
(2.3) the result of well-defined pointer arithmetic (7.6.6) using a safely-derived pointer value;
(2.4) the result of a well-defined pointer conversion (7.3.11, 7.6.3) of a safely-derived pointer value;
(2.5) the result of a \texttt{reinterpret\_cast} of a safely-derived pointer value;
(2.6) the result of a \texttt{reinterpret\_cast} of an integer representation of a safely-derived pointer value;
(2.7) 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 \texttt{std::intptr\_t} and it is one of the following:

(3.1) the result of a \texttt{reinterpret\_cast} of a safely-derived pointer value;
(3.2) the result of a valid conversion of an integer representation of a safely-derived pointer value;
(3.3) 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;
(3.4) the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value \(P\), if that result converted by \texttt{reinterpret\_cast<void*\>} would compare equal to a safely-derived pointer computable from \texttt{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 6.6.5. 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.

6.6.5.5 Duration of subobjects

The storage duration of subobjects and reference members is that of their complete object (6.6.2).

6.6.6 Alignment

Object types have alignment requirements (6.7.1, 6.7.2) 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 (9.11.2).
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; 
```
When `D` is the type of a complete object, it will have a subobject of type `B`, so it must be aligned appropriately for a `long double`. If `D` appears as a subobject of another object that also has `B` as a virtual base class, the `B` subobject might be part of a different subobject, reducing the alignment requirements on the `D` subobject. — end example] The result of the `alignof` operator reflects the alignment requirement of the type in the complete-object case.

An extended alignment is represented by an alignment greater than `alignof(std::max_align_t)` (17.2). It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (9.11.2). 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 `__STDCPP_DEFAULT_NEW_ALIGNMENT__` (15.10).

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

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

The alignment requirement of a complete type can be queried using an `alignof` expression (7.6.2.7). Furthermore, the narrow character types (6.7.1) 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 (9.11.2). — end note]

Comparing alignments is meaningful and provides the obvious results:

(7.1) — Two alignments are equal when their numeric values are equal.
(7.2) — Two alignments are different when their numeric values are not equal.
(7.3) — 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.

### 6.6.7 Temporary objects

Temporary objects are created

1. when a prvalue is converted to an xvalue (7.3.4),
2. when needed by the implementation to pass or return an object of trivially-copyable type (see below), and
3. when throwing an exception (14.1). [Note: The lifetime of exception objects is described in 14.1. — 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 (11.8) and whether it is deleted, for the constructor selected and for the destructor. However, in the special case of the operand of a `decltype-specifier` (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:

1. when binding a reference to a prvalue (9.3.3, 7.6.1.3, 7.6.1.6, 7.6.1.8, 7.6.1.10, 7.6.3),

§ 6.6.7
— 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} (9.3.4),
— 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).

— \textit{end note}  

\texttt{Example:} Consider the following code:

\begin{verbatim}
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);
}
\end{verbatim}

\texttt{X(2)} is constructed in the space used to hold \texttt{f()}’s argument and \texttt{Y(3)} is constructed in the space used to hold \texttt{g()}’s argument. Likewise, \texttt{f()}’s result is constructed directly in \texttt{b} and \texttt{g()}’s result is constructed directly in \texttt{c}. On the other hand, the expression \texttt{a = f(a)} requires a temporary for the result of \texttt{f(a)}, which is materialized so that the reference parameter of \texttt{A::operator=(const A&)} can bind to it.

— \textit{end example}  

3 When an object of class type \texttt{X} is passed to or returned from a function, if each copy constructor, move constructor, and destructor of \texttt{X} is either trivial or deleted, and \texttt{X} has at least one non-deleted copy or move constructor, 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 non-deleted 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). \textit{[Note:} This latitude is granted to allow objects of class type to be passed to or returned from functions in registers. \textit{— end note]}  

4 When an implementation introduces a temporary object of a class that has a non-trivial constructor (11.3.4.1, 11.3.4.2), 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 (11.3.6). Temporary objects are destroyed as the last step in evaluating the full-expression (6.8.1) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

5 There are three contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array with no corresponding initializer (9.3). 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, 11.3.4.2). 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 glvalue to which the reference is bound was obtained through one of the following:

- a temporary materialization conversion (7.3.4),
- (expression), where expression is one of these expressions,
- 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[]>{{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[]>{{1, 2, 3}[i]} : static_cast<int&&>(0);
// exactly one of the two temporaries is lifetime-extended
```

— end example] [Note: An explicit type conversion (7.6.1.3, 7.6.3) is interpreted as a sequence of elementary casts, covered above. [Example:

```cpp
const int& x = (const int&){1}; // temporary for value 1 has same lifetime as x
```

— end example] — end note] [Note: If a temporary object has a reference member initialized by another temporary object, lifetime extension applies recursively to such a member’s initializer. [Example:

```cpp
struct S {
    const int& m;
};
const S& s = S{i}; // both S and int temporaries have lifetime of s
```

— end example] — 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 (9.3) persists until the completion of the full-expression containing the expression-list.

- The lifetime of a temporary object bound to the returned value in a function return statement (8.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.

37) The same rules apply to initialization of an initializer_list object (9.3.4) with its underlying temporary array.
— A temporary bound to a reference in a new-initializer (7.6.2.5) persists until the completion of the full-expression containing the new-initializer. [Note: This may introduce a dangling reference. — end note] [Example:

```cpp
struct S {
  int mi; const std::pair<int, int>& mp;
};
S a { 1, {2,3} };  
S* p = new S{ 1, {2,3} };  // creates dangling reference
— end example]
```

7 The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary which is constructed earlier in the same full-expression. If the lifetime of two or more temporaries to which references are bound ends at the same point, these temporaries are destroyed at that point in the reverse order of the completion of their construction. In addition, the destruction of temporaries bound to references shall take into account the ordering of destruction of objects with static, thread, or automatic storage duration (6.6.5.1, 6.6.5.2, 6.6.5.3); that is, if `obj1` is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before `obj1` is destroyed; if `obj2` is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after `obj2` is destroyed.

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

6.7 Types [basic.types]

[Note: 6.7 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 (6.6.2), references (9.2.3.2), or functions (9.2.3.5). — end note]

2 For any object (other than a potentially-overlapping subobject) of trivially copyable type `T`, whether or not the object holds a valid value of type `T`, the underlying bytes (6.6.1) making up the object can be copied into an array of `char`, `unsigned char`, or `std::byte` (17.2.1). [8] If the content of that array is copied back into the object, the object shall subsequently hold its original value. [Example:

```cpp
#define N sizeof(T)
char buf[N];
T obj;
std::memcpy(buf, &obj, N);  // obj initialized to its original value
std::memmove(buf, &obj, N);  // between these two calls to std::memcpy, obj might be modified
std::memmove(&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 `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a potentially-overlapping subobject, if the underlying bytes (6.6.1) making up `obj1` are copied into `obj2`, `obj2` shall subsequently hold the same value as `obj1`. [Example:

```cpp
§ 6.7
63
```
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 T is the sequence of N unsigned char objects taken up by the object of type T, where N equals sizeof(T). The value representation of an object of type T is the set of bits that participate in representing a value of type T. Bits in the object representation that are not part of the value representation are padding bits. For trivially copyable types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.40

5 A class that has been declared but not defined, an enumeration type in certain contexts (9.6), or an array of unknown bound or of incomplete element type, is an incompletely-defined object type.41 Incompletely-defined object types and cv void are incomplete types (6.7.1). 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++; // ill-formed: X is incomplete
    arrp++; // ill-formed: 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; // ill-formed: different types
    xp++; // OK: X is complete
    arrp++; // ill-formed: 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 (6.7.1), enumeration types, pointer types, pointer-to-member types (6.7.2), std::nullptr_t, and cv-qualified (6.7.3) versions of these types are collectively called scalar types. Scalar types, trivially

40) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.
41) The size and layout of an instance of an incompletely-defined object type is unknown.
copyable class types (11.1), arrays of such types, and cv-qualified versions of these types are collectively called *trivially copyable types*. Scalar types, trivial class types (11.1), arrays of such types and cv-qualified versions of these types are collectively called *trivial types*. Scalar types, standard-layout class types (11.1), arrays of such types and cv-qualified versions of these types are collectively called *standard-layout types*.

A type is a *literal type* if it is:

1. possibly cv-qualified `void`; or
2. a scalar type; or
3. a reference type; or
4. an array of literal type; or
5. a possibly cv-qualified class type (Clause 11) that has all of the following properties:
   1. it has a trivial destructor,
   2. it is either a closure type (7.5.5.1), an aggregate type (9.3.1), or has at least one constexpr constructor or constructor template (possibly inherited (9.8) from a base class) that is not a copy or move constructor,
   3. if it is a union, at least one of its non-static data members is of non-volatile literal type, and
   4. 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 standard-layout class types (11.3).

### 6.7.1 Fundamental types [basic.fundamental]

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 size 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 (6.6.6) 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. — end example

The width of each signed integer type shall not be less than the values specified in Table 10. 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 (6.7) in the object representation are alternative representations of the value specified by the value representation. *Note:* Padding bits have unspecified value, but do not cause traps. See also ISO C 6.2.6.2. — end note

*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.

---

42) *This is also known as two’s complement representation.*


Table 10 — Minimum width

<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>

5 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.

6 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 (6.6.6), and range of representable values as the underlying type. Further, each value has the same representation in both types.

7 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 6.7. — end note]

8 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).

9 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>.

10 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]

11 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 (9.6) are not integral; however, unscoped enumerations can be promoted to integral types as specified in 7.3.6. — end note]

12 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.

13 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 (7.6.3). An expression of type cv void shall be used only as an expression statement (8.2), 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 (8.6.3) for a function with the return type cv void, or as the operand of an explicit conversion to type cv void.

§ 6.7.1 66
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]

### 6.7.2 Compound types

Compound types can be constructed in the following ways:

1. **arrays** of objects of a given type, 9.2.3.4;
2. **functions**, which have parameters of given types and return `void` or references or objects of a given type, 9.2.3.5;
3. **pointers to `cv void` or objects or functions (including static members of classes) of a given type**, 9.2.3.1;
4. **references** to objects or functions of a given type, 9.2.3.2. There are two types of references:
   - **lvalue reference**
   - **rvalue reference**
5. **classes** containing a sequence of objects of various types (Clause 11), a set of types, enumerations and functions for manipulating these objects (11.3.1), and a set of restrictions on the access to these entities (11.8);
6. **unions**, which are classes capable of containing objects of different types at different times, 11.4;
7. **enumerations**, which comprise a set of named constant values. Each distinct enumeration constitutes a different *enumerated type*, 9.6;
8. **pointers to non-static class members**, which identify members of a given type within objects of a given class, 9.2.3.3. Pointers to data members and pointers to member functions are collectively called *pointer-to-member* types.

These methods of constructing types can be applied recursively; restrictions are mentioned in 9.2.3.1, 9.2.3.4, 9.2.3.5, and 9.2.3.2. Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type `std::size_t` (17.2) is ill-formed.

The type of a pointer to `cv void` or a pointer to an object type is called an *object pointer type*. [Note: A pointer to `void` does not have a pointer-to-object type, however, because `void` is not an object type. —end note] The type of a pointer that can designate a function is called a *function pointer type*. A pointer to objects of type `T` is referred to as a “pointer to `T`”. [Example: A pointer to an object of type `int` is referred to as “pointer to `int`” and a pointer to an object of class `X` is called a “pointer to `X`”. —end example] Except for pointers to static members, text referring to “pointers” does not apply to pointers to members. Pointers to incomplete types are allowed although there are restrictions on what can be done with them (6.6.6). Every value of pointer type is one of the following:

1. **a pointer to an object or function** (the pointer is said to *point* to the object or function), or
2. **a pointer past the end of an object** (7.6.6), or
3. **the null pointer value** (7.3.11) for that type, or
4. **an invalid pointer value**.

A value of a pointer type that is a pointer to or past the end of an object *represents the address* of the first byte in memory (6.6.1) occupied by the object, 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 6.6.5. —end note] For purposes of pointer arithmetic (7.6.6) and comparison (7.6.9, 7.6.10), a pointer past the end of the last element of an array `x` of `n` elements is considered to be equivalent to a pointer to a hypothetical element `x[n]`. The value representation of pointer types is implementation-defined. Pointers to layout-compatible types shall have the same value representation and alignment requirements (6.6.6). [Note: Pointers to over-aligned

---

43) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.
44) For an object that is not within its lifetime, this is the first byte in memory that it will occupy or used to occupy.
Two objects $a$ and $b$ are pointer-interconvertible if:

1. they are the same object, or
2. one is a union object and the other is a non-static data member of that object (11.4), or
3. one is a standard-layout class object and the other is the first non-static data member of that object, or, if the object has no non-static data members, any base class subobject of that object (11.3), or
4. there exists an object $c$ such that $a$ and $c$ are pointer-interconvertible, and $c$ and $b$ are pointer-interconvertible.

If two objects are pointer-interconvertible, then they have the same address, and it is possible to obtain a pointer to one from a pointer to the other via a reinterpret_cast (7.6.1.9).  

A pointer to $cv$-qualified (6.7.3) or $cv$-unqualified 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*.

### 6.7.3 CV-qualifiers

A compound type (6.7.2) is not cv-qualified by the cv-qualifiers (if any) of the types from which it is compounded. Any cv-qualifiers applied to an array type affect the array element type (9.2.3.4).

There is a partial ordering on cv-qualifiers, so that a type can be said to be more cv-qualified than another. Table 11 shows the relations that constitute this ordering.

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

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 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

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.
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. An array type whose elements are cv-qualified is also considered to have the same cv-qualifications as its elements. [Example:

```c
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]

### 6.7.4 Integer conversion rank

Every integer type has an integer conversion rank defined as follows:

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.
2. The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.
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.
4. The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.
5. The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.
6. The rank of char shall equal the rank of signed char and unsigned char.
7. The rank of bool shall be less than the rank of all other standard integer types.
8. The ranks of char8_t, char16_t, char32_t, and wchar_t shall equal the ranks of their underlying types (6.7.1).
9. The rank of any extended signed integer type relative to another extended signed integer type with the same size is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
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 (7.2). —end note]

### 6.8 Program execution

#### 6.8.1 Sequential execution

An instance of each object with automatic storage duration (6.6.5.3) 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:

1. The constituent expression of an expression is that expression.
2. The constituent expressions of a braced-init-list or of a (possibly parenthesized) expression-list are the constituent expressions of the elements of the respective list.
3. The constituent expressions of a brace-or-equal-initializer of the form = initializer-clause are the constituent expressions of the initializer-clause.

[Example:

```c
struct A { int x; };
struct B { int y; struct A a; };
B b = { 5, { 1+1 } };  
```
The constituent expressions of the initializer used for the initialization of b are 5 and 1+1. —end example]

3. The immediate subexpressions of an expression e are
— the constituent expressions of e’s operands (7.2),
— any function call that e implicitly invokes,
— 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 e is a function call (7.6.1.2) or implicitly invokes a function, the constituent expressions of each default argument (9.2.3.6) used in the call, or
— if e creates an aggregate object (9.3.1), the constituent expressions of each default member initializer (11.3) used in the initialization.

4 A subexpression of an expression e is an immediate subexpression of e or a subexpression of an immediate subexpression of e. [Note: Expressions appearing in the compound-statement of a lambda-expression are not subexpressions of the lambda-expression. — end note]

5 A full-expression is

— an unevaluated operand (7.2),
— a constant-expression (7.7),
— an immediate invocation (7.7),
— an init-declarator (9.2) or a mem-initializer (11.9.2), including the constituent expressions of the initializer,
— an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object (6.6.7), or
— an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression.

If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression. For an initializer, performing the initialization of the entity (including evaluating default member initializers of an aggregate) is also considered part of the full-expression.

[Example:

```c
struct S {
  S(int i): I(i) { } // full-expression is initialization of I
  int& v() { return I; }
  ~S() noexcept(false) { }
private:
  int I;
};

S s1(1); // full-expression comprises call of S::S(int)
void f() {
  S s2 = 2; // full-expression comprises call of S::S(int)
  if (S(3).v()) // full-expression includes lvalue-to-rvalue and int to bool conversions, // performed before temporary is deleted at end of full-expression
    { }
  bool b = noexcept(S()); // exception specification of destructor of S considered for noexcept

    // full-expression is destruction of s2 at end of block
}
struct B {
  B(S = S(0));
};
B b[2] = { B(), B() }; // full-expression is the entire initialization // including the destruction of temporaries

— end example]

6 [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 (9.2.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — end note]
When calling a function (whether or not the function is inline), every value computation and side effect 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 \( A \). [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 §7.6.2.5. For another example, invocation of a conversion function (11.3.7.2) 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.

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

---

7 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 (6.8.2), 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.\

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 (6.6.1) 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 (6.8.2), 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++ + 1;
    // the behavior is undefined

    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 \( A \). [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 §7.6.2.5. For another example, invocation of a conversion function (11.3.7.2) 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

---

46 As specified in 6.6.7, 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.

47 In other words, function executions do not interleave with each other.
signal is received for another reason, the execution of the signal handler is usually unsequenced with respect to the rest of the program. —end note]

6.8.2 Multi-threaded executions and data races [intro.multithread]

1 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.48 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.

2 For a signal handler that is not executed as a result of a call to the std::raise function, it is unspecified which thread of execution contains the signal handler invocation.

6.8.2.1 Data races [intro.races]

1 The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [Note: 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]

2 Two expression evaluations conflict if one of them modifies a memory location (6.6.1) and the other one reads or modifies the same memory location.

3 The library defines a number of atomic operations (Clause 31) and operations on mutexes (Clause 32) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [Note: For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. “Relaxed” atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. —end note]

4 All modifications to a particular atomic object M occur in some particular total order, called the modification order of M. [Note: 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]

5 A release sequence headed by a release operation A on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is A, and every subsequent operation is an atomic read-modify-write operation.

6 Certain library calls synchronize with other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (31.4). [Note: Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. —end note] [Note: 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]

48) An object with automatic or thread storage duration (6.6.5) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (6.7.2).
An evaluation \( A \) carries a dependency to an evaluation \( B \) if

- 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
  - \( A \) 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);
  - \( A \) 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 a value written by any side effect in the release sequence headed 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 transitive 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 \( B \).
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]

The value of an atomic object $M$, as determined by evaluation $B$, shall be the value stored by some side effect $A$ that modifies $M$, where $B$ does not happen before $A$. [Note: The set of such side effects is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below. — end note]

If an operation $A$ that modifies an atomic object $M$ happens before an operation $B$ that modifies $M$, then $A$ shall be earlier than $B$ in the modification order of $M$. [Note: This requirement is known as write-write coherence. — end note]

If a value computation $A$ of an atomic object $M$ happens before a value computation $B$ of $M$, and $A$ takes its value from a side effect $X$ on $M$, then the value computed by $B$ shall either be the value stored by $X$ or the value stored by a side effect $Y$ on $M$, where $Y$ follows $X$ in the modification order of $M$. [Note: This requirement is known as read-read coherence. — end note]

If a value computation $A$ of an atomic object $M$ happens before an operation $B$ that modifies $M$, then $A$ shall take its value from a side effect $X$ on $M$, where $X$ precedes $B$ in the modification order of $M$. [Note: This requirement is known as read-write coherence. — end note]

If a side effect $X$ on an atomic object $M$ happens before a value computation $B$ of $M$, then the evaluation $B$ shall take its value from $X$ or from a side effect $Y$ that follows $X$ in the modification order of $M$. [Note: This requirement is known as write-read coherence. — end note]

[Note: The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — end note]

[Note: The value observed by a load of an atomic depends on the “happens before” relation, which depends on the values observed by loads of atomics. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the “happens before” relation derived as described above, satisfy the resulting constraints as imposed here. — end note]

Two actions are potentially concurrent if

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.

The execution of a program contains a data race if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior. [Note: It can be shown

§ 6.8.2.1
that programs that correctly use mutexes and `memory_order::seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as “sequential consistency”. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result 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 $B$.

--- end note

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]

### 6.8.2.2 Forward progress

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

1. **terminate,**
2. make a call to a library I/O function,
3. perform an access through a volatile glvalue, or
4. perform a synchronization operation or an atomic operation.

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

Executions of atomic functions that are either defined to be lock-free (31.9) or indicated as lock-free (31.5) are **lock-free executions**.

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. 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]

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

1. termination of the thread of execution,
2. performing an access through a volatile glvalue, or
3. completion of a call to a library I/O function, a synchronization operation, or an atomic operation.
An invocation of a standard library function that blocks (3.6) 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]

[Note: Because of this and the preceding requirement regarding what threads of execution have to perform eventually, it follows that no thread of execution can execute forever without an execution step occurring. — end note]

A thread of execution makes progress when an execution step occurs or a lock-free execution does not complete because there are other concurrent threads that are not blocked in a standard library function (see above).

For a thread of execution providing concurrent forward progress guarantees, the implementation ensures that the thread will eventually make progress for as long as it has not terminated. [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 (6.8.3.1) and the threads of execution created by std::thread (32.3.2) 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

§ 6.8.2.2
to strengthen a particular thread of execution whose too-weak progress guarantee is preventing overall progress. —end note]

18 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.

6.8.3 Start and termination

6.8.3.1 main function

A program shall contain a global function called main attached to the global module. Executing a program starts a main thread of execution (6.8.2, 32.3) in which the main function is invoked, and in which variables of static storage duration might be initialized (6.8.3.2) and destroyed (6.8.3.4). It is implementation-defined whether a program in a freestanding environment is required to define a main function. [Note: In a freestanding environment, start-up and termination is implementation-defined; start-up 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]

An implementation shall not redefine the main function. This function shall not be overloaded. Its type shall have C++ language linkage and it shall have a declared return type of type int, but otherwise its type is implementation-defined. An implementation shall allow both

(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 (9.2.3.5). In the latter form, for purposes of exposition, the first function parameter is called argc and the second function parameter is called argv, where argc shall be the number of arguments passed to the program from the environment in which the program is run. If argc is nonzero these arguments shall be supplied in argv[0] through argv[argc-1] as pointers to the initial characters of null-terminated multibyte strings (NTMBS) (16.4.2.2.5.2) 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 argc 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]

The function main shall not be used within a program. The linkage (6.5) 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 (9.4.4). The main function shall not be declared with a linkage-specification (9.10). 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]

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 (11.3.6). 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.

A return statement (8.6.3) 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 14.3).

6.8.3.2 Static initialization

Variables with static storage duration are initialized as a consequence of program initiation. Variables with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

Constant initialization is performed if a variable or temporary object with static or thread storage duration is initialized by a constant initializer (7.7) for the entity. If constant initialization is not performed, a variable with static storage duration (6.6.5.1) or thread storage duration (6.6.5.2) is zero-initialized (9.3). Together, zero-initialization and constant initialization are called static initialization; all other initialization is dynamic initialization. All static initialization strongly happens before (6.8.2.1) any dynamic initialization. [Note: The dynamic initialization of non-local variables is described in 6.8.3.3; that of local static variables is described in 8.7. —end note]
An implementation is permitted to perform the initialization of a variable with static or thread storage duration as a static initialization even if such initialization is not required to be done statically, provided that

(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

(3.2) 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 obj1 refers to an object obj2 of namespace scope potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified whether the value of obj2 used will be the value of the fully initialized obj2 (because obj2 was statically initialized) or will be the value of obj2 merely zero-initialized. For example,

```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 d1 is
    // dynamically initialized, or 1.0 otherwise

double d1 = fd(); // may be initialized statically or dynamically to 1.0
```
—end note]

6.8.3.3 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]

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

(2.1) If V and W have ordered initialization and V is defined before W within a single translation unit, the initialization of V is sequenced before the initialization of W.

(2.2) If V has partially-ordered initialization, W does not have unordered initialization, and V is defined before W in every translation unit in which W is defined, then

(2.2.1) if the program starts a thread (6.8.2) other than the main thread (6.8.3.1), the initialization of V strongly happens before the initialization of W;

(2.2.2) otherwise, the initialization of V is sequenced before the initialization of W.

(2.3) 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.

(2.4) 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 (6.2) 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 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,49 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:

49] A non-local variable with static storage duration having initialization with side effects is initialized in this case, even if it is not itself odr-used (6.2, 6.4.5.1).
It is implementation-defined whether either \texttt{a} or \texttt{b} is initialized before \texttt{main} is entered or whether the initializations are delayed until \texttt{a} is first odr-used in \texttt{main}. In particular, if \texttt{a} is initialized before \texttt{main} is entered, it is not guaranteed that \texttt{b} will be initialized before it is odr-used by the initialization of \texttt{a}, that is, before \texttt{A::A} is called. If, however, \texttt{a} is initialized at some point after the first statement of \texttt{main}, \texttt{b} will be initialized prior to its use in \texttt{A::A}. — end example]  

5 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 \texttt{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.

6 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 \texttt{t} is sequenced before the first non-initialization odr-use by \texttt{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.

7 If the initialization of a non-local variable with static or thread storage duration exits via an exception, the function \texttt{std::terminate} is called (14.5.1).

6.8.3.4 Termination  

1 Constructed objects (9.3) with static storage duration are destroyed and functions registered with \texttt{std::atexit} are called as part of a call to \texttt{std::exit} (17.5). The call to \texttt{std::exit} is sequenced before the destructions and the registered functions. [Note: Returning from \texttt{main} invokes \texttt{std::exit} (6.8.3.1). — 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 \texttt{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 \texttt{std::terminate} is called (14.5.1).
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 (6.8.2) 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>` 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 (9.6). Uses of overloaded operators are transformed into function calls as described in 12.5. 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 (12.5). — 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 12.3.1.2, 12.6.

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 vary 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 operands and the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.

---

50) The precedence of operators is not directly specified, but it can be derived from the syntax.
51) Overloaded operators are never assumed to be associative or commutative.
52) The cast and assignment operators must still perform their specific conversions as described in 7.6.3, 7.6.18 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 A glvalue is an expression whose evaluation determines the identity of an object, bit-field, or function.
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.
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).
4 An lvalue is a glvalue that is not an xvalue.
5 An rvalue is a prvalue or an xvalue.

Every expression belongs to exactly one of the fundamental classifications in this taxonomy: lvalue, xvalue, or prvalue. This property of an expression is called its value category. [Note: 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:

(a) the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type (7.6.1.2),
(b) a cast to an rvalue reference to object type (7.6.1.6, 7.6.3),
(c) a subscripting operation with an xvalue array operand (7.6.1.1),
(d) 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
(e) 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-prvalue (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 value reference to an lvalue is not such a context; see 9.3.3. — 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 9.3.3 and of temporaries in 6.6.7 indicates the behavior of lvalues and rvalues in other significant contexts.

9 Unless otherwise indicated (9.1.7.2), 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 (11.6.3). A prvalue 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. — 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 non-modifiable 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 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:53

- the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to the dynamic type of the object, or
- 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

If an expression initially has the type “reference to T” (9.2.3.2, 9.3.3), 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 6.6.3). — end note]

If a prvalue initially has the type “cv T”, where T is a cv-unqualified non-class, non-array type, the type of the expression is adjusted to T prior to any further analysis.

The cv-combined type of two types T1 and T2 is a type T3 similar to T1 whose cv-qualification signature (7.3.5) is:

- for every i > 0, cv\textsubscript{i} is the union of cv\textsubscript{i-1} and cv\textsubscript{i}2;
- if the resulting cv\textsubscript{i} is different from cv\textsubscript{i-1} or cv\textsubscript{i}2, then const is added to every cv\textsubscript{k} for 0 < k < i.

53) The intent of this list is to specify those circumstances in which an object may or may not be aliased.
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 \texttt{std::nullptr_t}, is:

\begin{enumerate}
\item if both \( p1 \) and \( p2 \) are null pointer constants, \texttt{std::nullptr_t};
\item if either \( p1 \) or \( p2 \) is a null pointer constant, \( T2 \) or \( T1 \), respectively;
\item if \( T1 \) or \( T2 \) is “pointer to \texttt{cv1 void}” and the other type is “pointer to \texttt{cv2} \( T \)”, where \( T \) is an object type or \texttt{void}, “pointer to \texttt{cv12 void}”, where \( \texttt{cv12} \) is the union of \texttt{cv1} and \texttt{cv2};
\item if \( T1 \) or \( T2 \) is “pointer to \texttt{noexcept} function” and the other type is “pointer to function”, where the function types are otherwise the same, “pointer to function”;\[\textit{NOTE}\text{: Given similar types \( T1 \) and \( T2 \), this construction ensures that both can be converted to \( T3 \).} - \textit{end note}\]
\item if \( T1 \) is “pointer to \texttt{cv1} \( C1 \)” and \( T2 \) is “pointer to \texttt{cv2} \( C2 \)”, where \( C1 \) is reference-related to \( C2 \) or \( C2 \) is reference-related to \( C1 \) (9.3.3), the \texttt{cv}-combined type of \( T1 \) and \( T2 \) or the \texttt{cv}-combined type of \( T2 \) and \( T1 \), respectively;
\item if \( T1 \) or \( T2 \) is “pointer to member of \( C1 \) of type function”, the other type is “pointer to member of \( C2 \) of type \texttt{noexcept} function”, and \( C1 \) is reference-related to \( C1 \) or \( C2 \) is reference-related to \( C1 \) (9.3.3), where the function types are otherwise the same, “pointer to member of \( C2 \) of type function” or “pointer to member of \( C1 \) of type function”, respectively;
\item if \( T1 \) is “pointer to member of \( C1 \) of type \texttt{cv1} \( U \)” and \( T2 \) is “pointer to member of \( C2 \) of type \texttt{cv2} \( U \)”, for some non-function type \( U \), where \( C1 \) is reference-related to \( C2 \) or \( C2 \) is reference-related to \( C1 \) (9.3.3), the \texttt{cv}-combined type of \( T2 \) and \( T1 \) or the \texttt{cv}-combined type of \( T1 \) and \( T2 \), respectively;
\item if \( T1 \) and \( T2 \) are similar types (7.3.5), the \texttt{cv}-combined type of \( T1 \) and \( T2 \);
\item otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.
\end{enumerate}

\[\textit{Example}:\]

\begin{verbatim}
typedef void *p;
typedef const int *q;
typedef int **pi;
typedef const int **pci;
\end{verbatim}

The composite pointer type of \( p \) and \( q \) is “pointer to \texttt{const void}”; the composite pointer type of \( \text{pi} \) and \( \text{pci} \) is “pointer to \texttt{const} pointer to \texttt{const int}”. - \textit{end example}\]

7.2.3 Context dependence

\[\text{expr.context}\]

1 In some contexts, unevaluated operands appear (7.5.7, 7.6.1.7, 7.6.2.4, 7.6.2.8, 9.1.7.2, Clause 13). An unevaluated operand is not evaluated. \[\text{NOTE}\text{: In an unevaluated operand, a non-static class member may be named (7.5.4) and naming of objects or functions does not, by itself, require that a definition be provided (6.2). An unevaluated operand is considered a full-expression (6.8.1).} - \textit{end note}\]

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-value 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:

\begin{enumerate}
\item \( \{\text{expression}\} \), where \text{expression} is one of these expressions,
\item \text{id-expression} (7.5.4),
\item subscripting (7.6.1.1),
\item class member access (7.6.1.4),
\item indirection (7.6.2.1),
\item pointer-to-member operation (7.6.4),
\item conditional expression (7.6.16) where both the second and the third operands are one of these expressions, or
\item comma expression (7.6.20) where the right operand is one of these expressions.
\end{enumerate}

\[\text{NOTE}\text{: Using an overloaded operator causes a function call; the above covers only operators with built-in meaning.} - \textit{end note}\text{] If the (possibly converted) expression is a prvalue, the temporary materialization...\]

\(\text{§ 7.2.3\] 84\)
A standard conversion sequence is a sequence of standard conversions in the following order:

1. Zero or one conversion from the following set: lvalue-to-rvalue conversion, array-to-pointer conversion, and function-to-pointer conversion.
2. Zero or one conversion from the following set: integral promotions, floating-point promotion, integral conversions, floating-point conversions, floating-integral conversions, pointer conversions, pointer-to-member conversions, and boolean conversions.
3. Zero or one function pointer conversion.
4. Zero or one qualification conversion.

A standard conversion sequence will be applied to an expression if necessary to convert it to a required destination type.

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

1. When used as operands of operators. The operator’s requirements for its operands dictate the destination type (7.6).
2. When used in the condition of an if statement (8.4.1) or iteration statement (8.5). The destination type is bool.
3. When used in the expression of a switch statement (8.4.2). The destination type is integral.
4. When used as the source expression for an initialization (which includes use as an argument in a function call and use as the expression in a return statement). The type of the entity being initialized is (generally) the destination type. See 9.3, 9.3.3.

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 (9.3).

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 (9.3).

Certain language constructs require conversion to a value having one of a specified set of types appropriate to the construct. An expression e of class type E 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: E 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 (9.2.3.2), 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.

For class types, user-defined conversions are considered as well; see 11.3.7. In general, an implicit conversion sequence (12.3.3.1) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. — end 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

1 A glvalue (7.2.1) of a non-function, non-array type \(T\) can be converted to a prvalue.\(^\text{54}\) 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 \(T\).\(^\text{55}\)

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

\[\begin{align*}
&\text{(2.1)} \quad \text{\(e\) is not potentially evaluated, or} \\
&\text{(2.2)} \quad \text{the evaluation of \(e\) results in the evaluation of a member \(ex\) of the set of potential results of \(e\), and \(ex\) names a variable \(x\) that is not odr-used by \(ex\) (6.2),}
\end{align*}\]

the value contained in the referenced object is not accessed. [Example:

\begin{verbatim}
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 due to access of x.n outside its lifetime
int n = g(true);  // OK, does not access y.n
\end{verbatim}
—end example]

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

\[\begin{align*}
&\text{(3.1)} \quad \text{If \(T\) is \cv std::nullptr_t, the result is a null pointer constant (7.3.11). [Note: Since the conversion does} \\
&\text{not access the object to which the glvalue refers, there is no side effect even if \(T\) is volatile-qualified (6.8.1),} \\
&\text{and the glvalue can refer to an inactive member of a union (11.4). — end note]}
\end{align*}\]

\[\begin{align*}
&\text{(3.2)} \quad \text{Otherwise, if \(T\) has a class type, the conversion copy-initializes the result object from the glvalue.}
\end{align*}\]

\[\begin{align*}
&\text{(3.3)} \quad \text{Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.6.5.4.2, 6.6.5.4.3),} \\
&\text{the behavior is implementation-defined.}
\end{align*}\]

\[\begin{align*}
&\text{(3.4)} \quad \text{Otherwise, the value contained in the object indicated by the glvalue is the prvalue result.}
\end{align*}\]

4 [Note: See also 7.2.1. — end note]

7.3.2 Array-to-pointer conversion

[conv.array]

1 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

[conv.func]

1 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.\(^\text{56}\)

2 [Note: See 12.4 for additional rules for the case where the function is overloaded. — end note]

7.3.4 Temporary materialization conversion

[conv.rval]

1 A prvalue of type \(T\) can be converted to an xvalue of type \(T\). This conversion initializes a temporary object (6.6.7) 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 11.3.6. — end note]

[Example:

\begin{verbatim}
struct X { int n; }
int k = X().n; // OK, X() prvalue is converted to xvalue
\end{verbatim}
—end example]

\[^{54}\text{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.}
\[^{55}\text{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.}
\[^{56}\text{This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.}
\]
7.3.5 Qualification conversions

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

\[
\text{"cv}\_0\ P_0\ cv\_1\ P_1\ \ldots\ cv\_{n-1}\ P_{n-1}\ cv_n\ U\text{" for } n \geq 0,
\]

where each \( cv_i \) is a set of cv-qualifiers (6.7.3), and each \( P_i \) is “pointer to” (9.2.3.1), “pointer to member of class \( C_i \) of type” (9.2.3.3), “array of \( N_i \)”, or “array of unknown bound of” (9.2.3.4). If \( P_i \) designates an array, the cv-qualifiers \( cv_{i+1} \) on the element type are also taken as the cv-qualifiers \( cv_i \) of the array. [Example: The type denoted by the *type-id const int** has two cv-decompositions, taking \( U \) as “int” and as “pointer to const int”. —end example] The \( n \)-tuple of cv-qualifiers after the first one in the longest cv-decomposition of \( T \), that is, \( cv_1, cv_2, \ldots, cv_n \), is called the cv-qualification signature of \( T \).

Two types \( T_1 \) and \( T_2 \) are *similar* if they have cv-decompositions with the same \( n \) such that corresponding \( P_i \) components are the same and the types denoted by \( U \) are the same.

A prvalue of type \( T_1 \) can be converted to type \( T_2 \) if the following conditions are satisfied, where \( cv_i^1 \) denotes the cv-qualifiers in the cv-qualification signature of \( T_j \).[57]

1. \( T_1 \) and \( T_2 \) are similar.
2. For every \( i > 0 \), if \( \text{const} \) is in \( cv_i^1 \) then \( \text{const} \) is in \( cv_i^2 \), and similarly for \( \text{volatile} \).
3. If the \( cv_i^1 \) and \( cv_i^2 \) are different, then \( \text{const} \) is in every \( cv_i^2 \) for \( 0 < k < i \).

[Note: If a program could assign a pointer of type \( T** \) to a pointer of type \( \text{const T**} \) (that is, if line #1 below were allowed), a program could inadvertently modify a \( \text{const} \) 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 example]

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 \( \mathbf{X} \) of type \( cv1\ T \)” can be converted to a prvalue of type “pointer to member of \( \mathbf{X} \) 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 (9.2.3.5). —end note]

7.3.6 Integral promotions

A prvalue of an integer type other than bool, char16_t, char32_t, or wchar_t whose integer conversion rank (6.7.4) 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.

A prvalue of type char16_t, char32_t, or wchar_t (6.7.1) 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.

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 (9.6): 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 (6.7.4) 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.

A prvalue of an unscoped enumeration type whose underlying type is fixed (9.6) 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

[57] These rules ensure that const-safety is preserved by the conversion.
A prvalue for an integral bit-field (11.3.9) can be converted to a prvalue of type \texttt{int} if \texttt{int} can represent all the values of the bit-field; otherwise, it can be converted to \texttt{unsigned int} if \texttt{unsigned int} can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

A prvalue of type \texttt{bool} can be converted to a prvalue of type \texttt{int}, with \texttt{false} becoming zero and \texttt{true} becoming one.

These conversions are called \textit{integral promotions}.

7.3.7 {Floating-point promotion} \[\text{conv.fpprom}\]

1 A prvalue of type \texttt{float} can be converted to a prvalue of type \texttt{double}. The value is unchanged.

2 This conversion is called \textit{floating-point promotion}.

7.3.8 {Integral conversions} \[\text{conv.integral}\]

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 \texttt{bool}, see 7.3.14. If the source type is \texttt{bool}, the value \texttt{false} is converted to zero and the value \texttt{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} \[\text{conv.double}\]

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} \[\text{conv.fpint}\]

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. \[\text{Note: If the destination type is \texttt{bool}, see 7.3.14. — end note}\]

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. \[\text{Note: Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating-point type. — end note}\] If the value being converted is outside the range of values that can be represented, the behavior is undefined. If the source type is \texttt{bool}, the value \texttt{false} is converted to zero and the value \texttt{true} is converted to one.

7.3.11 {Pointer conversions} \[\text{conv.ptr}\]

1 A \textit{null pointer constant} is an integer literal (5.13.2) with value zero or a prvalue of type \texttt{std::nullptr_t}. A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type (6.7.2) and is distinguishable from every other value of object pointer or function pointer type. Such a conversion is called a \textit{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 \texttt{std::nullptr_t}. \[\text{Note: The resulting prvalue is not a null pointer value. — end note}\]

2 A prvalue of type \texttt{“pointer to cv T”}, where \texttt{T} is an object type, can be converted to a prvalue of type \texttt{“pointer to cv void”}. The pointer value (6.7.2) is unchanged by this conversion.
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 (11.6) of D. If B is an inaccessible (11.8) or ambiguous (11.7) 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 (11.6) from B. If B is an inaccessible (11.8), ambiguous (11.7), or virtual (11.6.1) 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 B’s instance of D. Since the result has type “pointer to member of D of type cv T”, indirection through it with a D object is valid. The result is the same as if indirection through the pointer to member of B with the B subobject of D. The null member pointer value is converted to the null member pointer value of the destination type.

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(); private; };
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. If either operand is of scoped enumeration type (9.6), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.
2. If either operand is of type long double, the other shall be converted to long double.
3. Otherwise, if either operand is double, the other shall be converted to double.
4. Otherwise, if either operand is float, the other shall be converted to float.

58) 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, 11.6). 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*.

§ 7.4 89
— Otherwise, the integral promotions (7.3.6) shall be performed on both operands.\(^{59}\) 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.

(1.5.2) 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.

(1.5.3) 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.

(1.5.4) 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.

(1.5.5) Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

If one operand is of enumeration type and the other operand is of a different enumeration type or a floating-point type, this behavior is deprecated (D.1).

### 7.5 Primary expressions

#### 7.5.1 Literals

A literal is a primary expression. Its type depends on its form (5.13). A string literal is an lvalue; all other literals are prvalues.

#### 7.5.2 This

The keyword this names a pointer to the object for which a non-static member function (11.3.2.1) is invoked or a non-static data member’s initializer (11.3) 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:

```cpp
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 (11.3) of a class X, the expression this is a prvalue of type “pointer to X” within the optional default member initializer (11.3). It shall not appear elsewhere in the member-declarator.

The expression this shall not appear in any other context. [Example:

```cpp

```
class Outer {
    int a[sizeof(*this)];  // error: not inside a member function
    unsigned int sz = sizeof(*this);  // OK: in default member initializer

    void f() {
        int b[sizeof(*this)];  // OK
        struct Inner {
            int c[sizeof(*this)];  // error: not inside a member function of Inner
        }
    }
};
—end example

7.5.3 Parentheses

A parenthesized expression \((E)\) is a primary expression whose type, 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]

An id-expression that denotes an immediate function (9.1.5) shall appear as a subexpression of an immediate invocation or in an immediate function context (7.7).

An id-expression that denotes the specialization of a concept (13.6.8) results in a prvalue of type \texttt{bool}. The expression is \texttt{true} if the concept’s normalized constraint-expression (13.4.2) is satisfied (13.4.1) by the specified template arguments and \texttt{false} otherwise. [Example:
    template<typename T> concept C = true;
    static assert(C<int>);  // OK
—end example] [Note: A concept’s constraints are also considered when using a template name (13.2) and during overload resolution (Clause 12), and they are compared during the partial ordering of constraints (13.4.4). — end note]

A program that refers explicitly or implicitly to a function with a trailing requires-clause whose constraint-expression is not satisfied, other than to declare it, is ill-formed. [Example:
    void f(int) requires false;

60) This also applies when the object expression is an implicit \(*\text{this}\) (11.3.2).
void g() {
    f(0); // error: cannot call f
    void (*p1)(int) = f; // error: cannot take the address of f
    decltype(f)* p2 = nullptr; // error: the type decltype(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

### 7.5.4.1 Unqualified names

**unqualified-id:**

- identifier
- operator-function-id
- conversion-function-id
- literal-operator-id
- type-name
- decltype-specifier
- template-id

1 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 (9.2). [Note: For operator-function-ids, see 12.5; for conversion-function-ids, see 11.3.7.2; for literal-operator-ids, see 12.5.8; for template-ids, see 13.2. 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 (11.3.2). —end note]

2 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] If the entity is a template parameter object for a template parameter of type T (13.1), the type of the expression is const T. Otherwise, the type of the expression is the type of the result. [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 (9.5), 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:**

```c

nested-name-specifier template_opt unqualified-id
```

* `nested-name-specifier`:
  ```c
  ::
  type-name ::
  namespace-name ::
  decltype-specifier ::
  nested-name-specifier identifier ::
  nested-name-specifier template_opt simple-template-id ::
  ```

1 The type denoted by a decltype-specifier in a nested-name-specifier shall be a class or enumeration type.
A nested-name-specifier that denotes a class, optionally followed by the keyword `template` (13.2), and then followed by the name of a member of either that class (11.3) or one of its base classes (11.6), is a qualified-id; 6.4.3.1 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 (6.3.7). — end note] Where `type-name :: ~ type-name` is used, the two `type-name` 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`.

The nested-name-specifier :: names the global namespace. A nested-name-specifier that names a namespace (9.7), optionally followed by the keyword `template` (13.2), 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; 6.4.3.2 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 (9.5) and a prvalue otherwise.

A nested-name-specifier that denotes an enumeration (9.6), 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.

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

An `id-expression` that denotes the destructor of a type `T` names the destructor of `T` if `T` is a class type (11.3.6), otherwise the `id-expression` is said to name a pseudo-destructor.

If the `id-expression` names a pseudo-destructor, `T` shall be a scalar type and the `id-expression` shall appear as the right operand of a class member access (7.6.1.4) that forms the postfix-expression of a function call (7.6.1.2). [Note: Such a call has no effect. — end note]

[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-literal (5.13.8)
}
```
—end example]

### 7.5.5 Lambda expressions

A lambda-expression provides a concise way to create a simple function object. [Example:
#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); });
}

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]

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 9.2. — end note]

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 9.1.7.5. [Example:

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

A lambda is a generic lambda if there is a decl-specifier that is a placeholder-type-specifier in the decl-specifier-seq of a parameter-declaration of the lambda-expression, 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
```

7.5.5.1 Closure types

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.

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 (6.4.2). 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 (9.3.1). 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:

1. the size and/or alignment of the closure type,
2. whether the closure type is trivially copyable (11.1), or
3. whether the closure type is a standard-layout class (11.1).

An implementation shall not add members of rvalue reference type to the closure type.

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) (12.5.4) whose parameters and return type are described by the lambda-expression’s parameter-declaration-clause and trailing-return-type respectively, and whose template-parameter-list consists of the specified template-parameter-list, if any. The requires-clause of the function call operator template is the requires-clause immediately following <template-parameter-list>, if any. The trailing requires-clause of the function call operator template is the requires-clause following the lambda-declarator, if any. [Note: The function call operator for a generic lambda might be an abbreviated function template (9.2.3.5). — end note] [Example:

```c++
auto glambda = [] (auto a, auto&& b) { return a < b; }; // OK
auto vglambda = [] (auto printer) {
    return [=](auto&& ... ts) {
        printer(std::forward<decltype(ts)>)(ts)...); // OK: ts is a function parameter pack
```
The function call operator or operator template is declared `const` (11.3.2) 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 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 (9.1.5). It is an immediate function (9.1.5) 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; };
static_assert(ID(3) == 3); // OK

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

```
4
```

```
5
```

```
6
```

§ 7.5.5.1
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 (9.10) 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.

8 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.

9 [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

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
```
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 shall 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 (14.4). [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).
auto NC = [] (auto a) { static int s; return a; };
static_assert(Fwd(NC,3) == 3); // ill-formed
```

— end example]

The lambda-expression’s compound-statement yields the function-body (9.4) of the function call operator, but for purposes of name lookup (6.4), determining the type and value of this (11.3.2.1) and transforming id-expressions referring to non-static class members into class member access expressions using (*this) (11.3.2), 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 9.4.1.

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 (11.3.4.2). It has a deleted copy assignment operator if the lambda-expression has a lambda-capture and defaulted copy and move assignment operators otherwise (11.3.5). [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 (11.3.6).

A member of a closure type shall not be explicitly instantiated (13.8.2), explicitly specialized (13.8.3), or named in a friend declaration (11.8.3).

### 7.5.5.2 Captures

```cpp
lambda-capture:
capture-default
capture-list
capture-default , capture-list
capture-default:
&
```
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 &, no identifier in a simple-capture of that lambda-capture shall be preceded by &. If a lambda-capture includes a capture-default that is =, each simple-capture of that lambda-capture shall be of the form "& identifier", "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
  [&], 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
}
```

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 (6.3.3) or it appears within a default member initializer and its innermost enclosing scope is the corresponding class scope (6.3.7).

The identifier in a simple-capture is looked up using the usual rules for unqualified name lookup (6.4.1); 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
}
```

An init-capture 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 = [&r = x, x = x+1]()->int {
    r += 2;
    return x+2;
}(); // Updates x to 6, and initializes y to 7.

auto z = [a = 42](int a) { return 1; }; // error: parameter and local variable have the same name

--- end example

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 (8.4.1). [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]

--- end example

An entity is captured if it is captured explicitly or implicitly. An entity captured by a lambda-expression is odr-used (6.2) 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 (6.2). — 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];
    x[0][0] = i;
    // OK: i is explicitly captured by m2 and implicitly captured by m1
  };
};
struct s1 {
  int f;
  void work(int n) {
    int m = n*n;
    int j = 40;
    auto m3 = [this,m] {
      auto m4 = [&,j] {
        int x = n;  // error: j 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] { };  // error: *this not captured by outer lambda-expression
    }();
  }
};
—end example

9 [Note: Because local entities are not odr-usable within a default argument (6.2), 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 (9.2.3.6). —end note] [Example:

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

10 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.

11 Every id-expression within the compound-statement of a lambda-expression that is an odr-use (6.2) 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. — 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:

```cpp
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]
```

12 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:

```cpp
// The inner closure type must be a literal type regardless of how reference captures are represented.
static_assert([](int n) { return [&n] { return ++n; }();)(3) == 4);
—end example] A bit-field or a member of an anonymous union shall not be captured by reference.

13 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:

```cpp
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]
```

14 If a lambda-expression m2 captures an entity and that entity is captured by an immediately enclosing lambda-expression m1, then m2’s capture is transformed as follows:

(14.1) if m1 captures the entity by copy, m2 captures the corresponding non-static data member of m1’s closure type;

(14.2) if m1 captures the entity by reference, m2 captures the same entity captured by m1.

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

```cpp
int a = 1, b = 1, c = 1;
auto m1 = [a, &b, &c] () mutable {
    auto m2 = [a, b, &c] () mutable {
        std::cout << a << b << c;
        a = 4; b = 4; c = 4;
    };
    a = 3; b = 3; c = 3;
    m2();
};
a = 2; b = 2; c = 2;
m1();
std::cout << a << b << c;
```
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]

[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]

A simple-capture followed by an ellipsis is a pack expansion (13.6.3). An init-capture preceded by an ellipsis is a pack expansion that introduces an init-capture pack (13.6.3) whose declarative region is the lambda-expression’s compound-statement. [Example:

```cpp
template<class... Args>
void f(Args... args) {
    auto lm = [&args...] { return g(args...); };
    lm();

    auto lm2 = [...xs=std::move(args)] { return g(xs...); };
    lm2();
}
```

— end example]

### 7.5.6 Fold expressions

A fold expression performs a fold of a pack (13.6.3) 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

```
+ - * / % ^ & | << >>
+= -= == /= %= ^= &= |= <<= >>= =
== <! > <= >= && || , .* ->*
```

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 (13.6.3).

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:

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

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

— end example]

### 7.5.7 Requires expressions

A requires-expression provides a concise way to express requirements on template arguments that can be checked by name lookup (6.4) 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

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).

[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} -> const typename T::type&;
};
```

A requires-expression can also be used in a requires-clause (Clause 13) 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]

A requires-expression may introduce local parameters using a parameter-declaration-clause (9.2.3.5). 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]

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.

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
   };
```
7.5.7.1 Simple requirements

A simple-requirement asserts the validity of an expression. [Note: The enclosing requires-expression will evaluate to false if substitution of template arguments into the expression fails. The expression is an unevaluated operand (7.2). — end note] [Example:

```cpp
template<typename T> concept C =
    requires (T a, T b) {
        a + b; // C<T> is true if a + b is a valid expression
    };
```
— end example]

7.5.7.2 Type requirements

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

```cpp
template<typename T, typename T::type = 0> struct S;
template<typename T> using Ref = T&;
template<typename T> concept C = requires {
    typename T::inner; // required nested member name
    typename S<T>; // required class template specialization
    typename Ref<T>; // required alias template substitution, fails if T is void
};
```
— end example]

2 A type-requirement that names a class template specialization does not require that type to be complete (6.7).

7.5.7.3 Compound requirements

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

1. Substitution of template arguments (if any) into the expression is performed.
2. If the noexcept specifier is present, E shall not be a potentially-throwing expression (14.4).
3. If the return-type-requirement is present, then:
   1.4.1 Substitution of template arguments (if any) into the return-type-requirement is performed.
   1.4.2 If the return-type-requirement is a trailing-return-type (9.2), E is implicitly convertible to the type named by the trailing-return-type. If conversion fails, the enclosing requires-expression is false.
   1.4.3 If the return-type-requirement is of the form -> type-constraint, then the contextually-determined type being constrained is decltype((E)). The immediately-declared constraint (Clause 13) of decltype((E)) shall be satisfied. [Example: Given concepts C and D,

```cpp
    requires {
        { E1 } -> C;
        { E2 } -> D<A_1, ···, A_n>;
    };
```

is equivalent to

```cpp
    requires {
        E1; requires C<decltype((E1))>;
```
E2; requires D<decltype((E2)), A1, ..., An>; }

(including in the case where n is zero). —end example]

2 [Example:
    template<typename T> concept C1 = requires(T x) {
        {x++};
    }
    The compound-requirement in C1 requires that x++ is a valid expression. It is equivalent to the simple-requirement x++;.
    template<typename T> concept C2 = requires(T x) {
        {*x} -> typename T::inner;
    }
    The compound-requirement in C2 requires that *x is a valid expression, that typename T::inner is a valid type, and that *x is implicitly convertible to typename T::inner.
    template<typename T> concept C3 = requires(T x) {
        {g(x)} noexcept;
    }
    The compound-requirement in C3 requires that g(x) is a valid expression and that g(x) is non-throwing. —end example]

7.5.7.4 Nested requirements [expr.prim.req.nested]

    nested-requirement:
    requires constraint-expression ;

1 A nested-requirement can be used to specify additional constraints in terms of local parameters. The constraint-expression shall be satisfied (13.4.2) 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 13.4.2. [Example:
    template<typename U> concept C = sizeof(U) == 1;
    template<typename T> concept D = requires (T t) {
        requires C<decltype (+t)>
    };
    D<T> is satisfied if sizeof(decltype (+t)) == 1 (13.4.1.2). —end example]

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

1 Postfix expressions group left-to-right.
postfix-expression:
  primary-expression
  postfix-expression [ expr-or-braced-init-list ]
  postfix-expression ( expression-list_opt )
  simple-type-specifier ( expression-list_opt )
  typename-specifier ( expression-list_opt )
  simple-type-specifier braced-init-list
  typename-specifier braced-init-list
  postfix-expression . template_opt id-expression
  postfix-expression -> template_opt id-expression
  postfix-expression ++
  postfix-expression --
  dynamic_cast < type-id > ( expression )
  static_cast < type-id > ( expression )
  reinterpret_cast < type-id > ( expression )
  const_cast < type-id > ( expression )
  typeid ( expression )
  typeid ( type-id )

expression-list:
  initializer-list

2 [Note: The > token following the type-id in a dynamic_cast, static_cast, reinterpret_cast, or const_cast may be the product of replacing a >> token by two consecutive > tokens (13.2). — end note]

7.6.1.1 Subscripting [expr.sub]

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.61 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 \( E2 \).

2 [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 9.2.3.4 for details of array types. — end note]

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

7.6.1.2 Function call [expr.call]

A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of initializer-clause which constitute the arguments to the function. 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 (11.3.2, 11.3.8) 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 11.3.2). — end note]

3 If a function or member function name is used, the appropriate function and the validity of the call are determined according to the rules in 12.3. 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 (11.6.2) 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. 11.9.4 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 (6.4) does not find a declaration of that name, the program is ill-formed. No function is implicitly declared by such a call. — end note]

61) This is true even if the subscript operator is used in the following common idiom: \( kx[0] \).
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, the function call has no effect.

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 (9.10).

When a function is called, each parameter (9.2.3.5) is initialized (9.3, 11.3.4.2) with its corresponding argument. If there is no corresponding argument, the default argument for the parameter is used; the program is ill-formed if one is not present. [Example:

```cpp
template<typename ...T> int f(int n = 0, T ...t);
int x = f<int>();
// error: no argument for second function parameter
```
—end example] If the function is a non-static member function, the this parameter of the function (11.3.2.1) is initialized with a pointer to the object of the call, converted as if by an explicit type conversion (7.6.3). [Note: There is no access or ambiguity checking on this conversion; the access checking and disambiguation are done as part of the (possibly implicit) class member access operator. See 11.7, 11.8.2, and 7.6.1.4. —end note] When a function is called, the type of any parameter shall not be a class type that is either incomplete or abstract. [Note: This still allows a parameter to be a pointer or reference to such a type. However, it prevents a passed-by-value parameter to have an incomplete or abstract class type. —end note] It is implementation-defined whether the lifetime of a parameter ends when the function in which it is defined returns or at the end of the enclosing full-expression. The initialization and destruction of each parameter occurs within the context of the calling function. [Example: The access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a function-try-block (Clause 14) with a handler that could handle the exception, this handler is not considered. —end example]

The postfix-expression is sequenced before each expression in the expression-list and any default argument. The initialization of a parameter, including every associated value computation and side effect, is indeterminately sequenced with respect to that of any other parameter. [Note: All side effects of argument evaluations are sequenced before the function is entered (see 6.8.1). —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 12.3.1.2. —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 operand of the evaluated return statement (8.6.3) in 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 (9.2.3.2); 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 (9.1.7, 5.13, 5.13.5, 9.2.3.4, 6.6.7). 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 (9.2.3.6)) or more arguments (by using the ellipsis, ..., or a function parameter pack (9.2.3.5)) than the number of parameters in the function definition (9.4). [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 (13.6.3), thus each such argument has a corresponding parameter when a function template specialization is actually called. — end note] 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 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 class type (Clause 11) having a non-trivial copy constructor, a non-trivial move constructor, or a non-trivial destructor, 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 (6.8.3.1).

A function call is an lvalue if the result type is an lvalue reference type or an rvalue reference to function type, an rvalue if the result type is an rvalue reference to object type, and a prvalue otherwise.

### 7.6.1.3 Explicit type conversion (functional notation)

A simple-type-specifier (9.1.7.2) or typename-specifier (13.7) 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 (12.3.1.8) 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 (9.3) with (after pack expansion, if any), the expression is a prvalue of the specified type that performs no initialization.

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). — end note]

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 (6.3.7). — 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: 6.4.5 describes how names are looked up after the . and -> operators. — end note]

---

62) 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.

63) Note that *(E1) is an lvalue.
5 If \( E_2 \) is a bit-field, \( E_1.E_2 \) is a bit-field. The type and value category of \( E_1.E_2 \) are determined as follows. In the remainder of 7.6.1.4, \( eq \) represents either \( \text{const} \) or the absence of \( \text{const} \) and \( vq \) represents either \( \text{volatile} \) or the absence of \( \text{volatile} \). \( cv \) represents an arbitrary set of \( cv \)-qualifiers, as defined in 6.7.3.

6 If \( E_2 \) is declared to have type “reference to \( T \)”, then \( E_1.E_2 \) is an lvalue; the type of \( E_1.E_2 \) is \( T \). Otherwise, one of the following rules applies.

(6.1) — If \( E_2 \) is a static data member and the type of \( E_2 \) is \( T \), then \( E_1.E_2 \) is an lvalue; the expression designates the named member of the class. The type of \( E_1.E_2 \) is \( T \).

(6.2) — If \( E_2 \) is a non-static data member and the type of \( E_1 \) is \( "eq_{1} \ vq_{1} \\ T" \), and the type of \( E_2 \) is \( "eq_{2} \ vq_{2} \\ T" \), the expression designates the named member of the object designated by the first expression. If \( E_1 \) is an lvalue, then \( E_1.E_2 \) is an lvalue; otherwise \( E_1.E_2 \) is an xvalue. Let the notation \( vq_{12} \) stand for the “union” of \( vq_{1} \) and \( vq_{2} \); that is, if \( vq_{1} \) or \( vq_{2} \) is \( \text{volatile} \), then \( vq_{12} \) is \( \text{volatile} \). 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 \( \text{const} \), then \( cq_{12} \) is \( \text{const} \).

If \( E_2 \) is declared to be a \( \text{mutable} \) member, then the type of \( E_1.E_2 \) is \( "vq_{12} \\ T" \). If \( E_2 \) is not declared to be a \( \text{mutable} \) member, then the type of \( E_1.E_2 \) is \( "cq_{12} \ vq_{12} \\ T" \).

(6.3) — If \( E_2 \) is a (possibly overloaded) member function, function overload resolution (12.3) is used to determine whether \( E_1.E_2 \) refers to a static or a non-static member function.

(6.3.1) — If it refers to a static member function and the type of \( E_2 \) is “function of parameter-type-list returning \( T \)”, then \( E_1.E_2 \) is an lvalue; the expression designates the static member function. The type of \( E_1.E_2 \) is the same type as that of \( E_2 \), namely “function of parameter-type-list returning \( T \)”,

(6.3.2) — Otherwise, if \( E_1.E_2 \) refers to a non-static member function and the type of \( E_2 \) is “function of parameter-type-list \( cv \ vq_{12} \text{ref-qualifier}_{opt} \) returning \( T \)”, then \( E_1.E_2 \) is a prvalue. The expression designates a non-static member function. The expression can be used only as the left-hand operand of a member function call (11.3.1). [Note: Any redundant set of parentheses surrounding the expression is ignored (7.5.3). — end note] The type of \( E_1.E_2 \) is “function of parameter-type-list \( cv \) returning \( T \)”,

(6.4) — If \( E_2 \) is a nested type, the expression \( E_1.E_2 \) is ill-formed.

(6.5) — If \( E_2 \) is a member enumerator and the type of \( E_2 \) is \( T \), the expression \( E_1.E_2 \) is a prvalue. The type of \( E_1.E_2 \) is \( T \).

7 If \( E_2 \) is a non-static data member or a non-static member function, the program is ill-formed if the class of which \( E_2 \) is directly a member is an ambiguous base (11.7) of the naming class (11.8.2) of \( E_2 \). [Note: The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 11.8.2. — end note]

7.6.1.5 Increment and decrement

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 \( \text{cv bool} \), or a pointer to a complete object type. The value of the operand object is modified 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-sequence function call, the operation of postfix \( ++ \) is a single evaluation. [Note: Therefore, a function call shall not 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

The result of the expression \( \text{dynamic\_cast}<\text{T}>(v) \) is the result of converting the expression \( v \) to type \( T \). \( T \) shall be a pointer or reference to a complete class type, or “pointer to \( \text{cv void} \)”. The \( \text{dynamic\_cast} \) operator shall not cast away constness (7.6.1.10).

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 \( T \).
If the type of \( v \) is the same as \( T \) (ignoring cv-qualifications), the result is \( v \) (converted if necessary).

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 \( v \).  

[Example:

```cpp
struct B { }
struct D : B { }
void foo(D* dp) {
  B* bp = dynamic_cast<B*>(dp);  // equivalent to B* bp = dp;
}
— end example]
```

Otherwise, \( v \) shall be a pointer to or a glvalue of a polymorphic type (11.6.2).

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 (14.1) of a type that would match a handler (14.3) of type std::bad_cast (17.7.3).

[Example:

```cpp
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; // ill-formed: cast from virtual base
  E* ep1 = dynamic_cast<E*>(ap); // succeeds
}
— end example]  

[Note: Subclause 11.9.4 describes the behavior of a `dynamic_cast` applied to an object under construction or destruction. — end note]

64) The most derived object (6.6.2) pointed or referred to by \( v \) can contain other \( B \) objects as base classes, but these are ignored.
7.6.1.7 Type identification

1 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.

2 When typeid is applied to a glvalue whose type is a polymorphic class type (11.6.2), the result refers to a std::type_info object representing the type of the most derived object (6.6.2) (that is, the dynamic type) to which the glvalue refers. If the glvalue is obtained by applying the unary * operator to a pointer66 and the pointer is a null pointer value (7.3.11), the typeid expression throws an exception (14.1) of a type that would match a handler of type std::bad_typeid exception (17.7.4).

3 When typeid is applied to an expression other than a glvalue of a polymorphic class type, the result refers to a std::type_info object representing the static type of the expression. Lvalue-to-rvalue (7.3.1), 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).

4 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.

5 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:

```cpp
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]

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

[Note: Subclause 11.9.4 describes the behavior of typeid applied to an object under construction or destruction. —end note]

7.6.1.8 Static cast

1 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).

2 An lvalue of type “cv1 B”, where B is a class type, can be cast to type “reference to cv2 D”, where D is a class derived (11.6) 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. An xvalue of type “cv1 B” can be cast to type “rvalue reference to cv2 D” with the same constraints as for an lvalue of type “cv1 B”. If the object of type “cv1 B” is actually a base class subobject of an object of type D, the result refers to the enclosing object of type D. Otherwise, the behavior is undefined. [Example:

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

65) The recommended name for such a class is extended_type_info.
66) If p is an expression of pointer type, then *p, *(p), (p), (**p), *(*(p)), and so on all meet this requirement.
A prvalue of type “pointer to
A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the
The lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and function-to-pointer (7.3.3) conversions are applied
The inverse of any standard conversion sequence (7.3) not containing an lvalue-to-rvalue (7.3.1), array-to-
An expression e can be explicitly converted to a type T if there is an implicit conversion sequence (12.3.3.1)
from e to T, or if overload resolution for a direct-initialization (9.3) of an object or reference of type T from e
would find at least one viable function (12.3.2). If T is a reference type, the effect is the same as performing
the declaration and initialization
T t(e);
for some invented temporary variable t (9.3) 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
attemping to convert an expression of class type to an inaccessible or ambiguous base class. — end note]
Otherwise, the static_cast shall perform one of the conversions listed below. No other conversion shall be
performed explicitly using a static_cast.
Any expression can be explicitly converted to type cv void, in which case it becomes a discarded-value
expression (7.2). [Note: However, if the value is in a temporary object (6.6.7), the destructor for that object
is not executed until the usual time, and the value of the object is preserved for the purpose of executing the
destructor. — end note]
The inverse of any standard conversion sequence (7.3) not containing an lvalue-to-rvalue (7.3.1), array-to-
pointer (7.3.2), function-to-pointer (7.3.3), null pointer (7.3.11), null member pointer (7.3.12), boolean (7.3.14),
or function pointer (7.3.13) 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:

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

— end example]

An lvalue of type “cv1 T1” can be cast to type “rvalue reference to cv2 T2” if “cv2 T2” is reference-compatible
with “cv1 T1” (9.3.3). 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 (11.8) or ambiguous (11.7) base class of T1, a program that necessitates such a cast is ill-formed.

A value of a scoped enumeration type (9.6) can be explicitly converted to an integral type; the result is the
same as converting the original value to the underlying type of
The null pointer value (7.3.11) is converted to the null pointer value of the destination type. If the prvalue of
a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as
that of converting to the enumeration’s underlying type and then to the destination type. A value of
a scoped enumeration type can also be explicitly converted to a floating-point type; the result is the same as
that of converting from the original value to the floating-point type.

A value of integral or enumeration type can be explicitly converted to a complete enumeration type. If the
enumeration type has a fixed underlying type, the value is first converted to that type by integral conversion,
if necessary, and then to the enumeration type. If the enumeration type does not have a fixed underlying
type, the value is unchanged if the original value is within the range of the enumeration values (9.6), 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.

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 (11.6) 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 (7.3.11) 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.
A prvalue of type “pointer to member of D of type cvf T” can be converted to a prvalue of type “pointer to member of B of type cvf T”, where D is a complete class type and B is a base class (11.6) of D, if cvf is the same cv-qualification as, or greater cv-qualification than, cvf.67 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 cvf void” can be converted to a prvalue of type “pointer to cvf T”, where T is an object type and cvf is the same cv-qualification as, or greater cv-qualification than, cvf. 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 (6.7.2) with a, the result is a pointer to b. Otherwise, the pointer value is unchanged by the conversion. [Example:

```c
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 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 expression v. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

1. 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.

2. [Note: The mapping performed by `reinterpret_cast` might, or might not, produce a representation different from the original value. — end note]

3. A pointer can be explicitly converted to any integral type large enough to hold all values of its type. The mapping function is implementation-defined. [Note: 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]

4. A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined. [Note: Except as described in 6.6.5.4.3, the result of such a conversion will not be a safely-derived pointer value. — end note]

5. A function pointer can be explicitly converted to a function pointer of a different type. [Note: The effect of calling a function through a pointer to a function type (9.2.3.5) that is not the same as the type used in the definition of the function is undefined. — 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]

6. An object pointer can be explicitly converted to an object pointer of a different type.68 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

---

67 Function types (including those used in pointer-to-member-function types) are never cv-qualified; see 9.2.3.5.

68 The types may have different cv-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.
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 (7.3.11) 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]

A prvalue of type “pointer to member of X of type T1” can be explicitly converted to a prvalue of a different type “pointer to member of Y of type T2” if T1 and T2 are both function types or both object types. The null 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:

1. Converting a prvalue of type “pointer to member function” to a different pointer-to-member-function type and back to its original type yields the original pointer-to-member value.
2. Converting a prvalue of type “pointer to data member of X of type T1” to the type “pointer to data member of Y of type T2” (where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer-to-member value.

A glvalue of type T1, designating an object x, can be cast to the type “reference to T2” if an expression of type “pointer to T1” can be explicitly converted to the type “pointer to T2” using a reinterpret_cast. The result is that of reinterpret_cast<T2 *>(p) where p is a pointer to x of type “pointer to T1”. No temporary is created, no copy is made, and no constructors (11.3.4) or conversion functions (11.3.7) are called.70

7.6.1.10 Const cast [expr.const.cast]

The result of the expression const_cast<T>(v) is of type T. If T is an lvalue reference to object type, the result is an lvalue; if T is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue and the lvalue-to-rvalue (7.3.1), 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 const_cast are listed below. No other conversion shall be performed explicitly using const_cast.

[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 T1 and T2 (7.3.5), a prvalue of type T1 may be explicitly converted to the type T2 using a const_cast. The result of a const_cast refers to the original entity. [Example:

```cpp
typedef int *A[3]; // array of 3 pointer to int
typedef const int *const CA[3]; // array of 3 const pointer to const int

CA &&r = A[0]; // OK, reference binds to temporary array object after qualification conversion to type CA
A &&r1 = const_cast<A>(CA[0]); // error: temporary array decayed to pointer
A &&r2 = const_cast<A&>(CA[0]); // OK
```

— end example]

For two object types T1 and T2, if a pointer to T1 can be explicitly converted to the type “pointer to T2” using a const_cast, then the following conversions can also be made:

1. an lvalue of type T1 can be explicitly converted to an lvalue of type T2 using the cast const_cast<T2&>;
2. a glvalue of type T1 can be explicitly converted to an xvalue of type T2 using the cast const_cast<T2&&>;
3. if T1 is a class type, a prvalue of type T1 can be explicitly converted to an xvalue of type T2 using the cast const_cast<T2&&>.

69 T1 and T2 may have different cv-qualifiers, subject to the overall restriction that a reinterpret_cast cannot cast away constness.

70 This is sometimes referred to as a type pun when the result refers to the same object as the source glvalue.
The result of a reference \texttt{const\_cast} refers to the original object if the operand is a glvalue and to the result of applying the temporary materialization conversion (7.3.4) otherwise.

5 A null pointer value (7.3.11) 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 \texttt{const\_cast} that casts away a const-qualifier\textsuperscript{71} may produce undefined behavior (9.1.7.1). —end note]

7 A conversion from a type \texttt{T1} to a type \texttt{T2} casts away constness if \texttt{T1} and \texttt{T2} are different, there is a cv-decomposition (7.3.5) of \texttt{T1} yielding \texttt{n} such that \texttt{T2} has a cv-decomposition of the form

\[
\texttt{cv}_0 \ P_0^2 \ \texttt{cv}_1^2 \ P_1^2 \ \cdots \ \texttt{cv}_{n-1}^2 \ P_{n-1}^2 \ \texttt{cv}_n^2 \ U_2,
\]

and there is no qualification conversion that converts \texttt{T1} to

\[
\texttt{cv}_0^2 \ P_0^1 \ \texttt{cv}_1^1 \ P_1^1 \ \cdots \ \texttt{cv}_{n-1}^1 \ P_{n-1}^1 \ \texttt{cv}_n^1 \ U_1.
\]

8 Casting from an lvalue of type \texttt{T1} to an lvalue of type \texttt{T2} using an lvalue reference cast or casting from an expression of type \texttt{T1} to an xvalue of type \texttt{T2} using an rvalue reference cast casts away constness if a cast from a prvalue of type “pointer to \texttt{T1}” to the type “pointer to \texttt{T2}” casts away constness.

9 [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.

\texttt{unary-expression:}

- \texttt{postfix-expression}
- \texttt{unary-operator cast-expression}
- \texttt{++ cast-expression}
- \texttt{-- cast-expression}
- \texttt{await-expression}
- \texttt{sizeof unary-expression}
- \texttt{sizeof ( type-id )}
- \texttt{sizeof ( identifier )}
- \texttt{alignof ( type-id )}
- \texttt{noexcept-expression}
- \texttt{new-expression}
- \texttt{delete-expression}

\texttt{unary-operator:} one of

\texttt{*} \& + - ! ~

7.6.2.1 Unary operators

\texttt{unary-operator:} one of \texttt{*} \& + - ! ~

1 The unary \texttt{*} operator performs \texttt{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 \texttt{T}”, the type of the result is “\texttt{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]

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

3 The result of the unary \texttt{&} operator is a pointer to its operand. The operand shall be an lvalue or a \texttt{qualified-id}. If the operand is a \texttt{qualified-id} naming a non-static or variant member \texttt{m} of some class \texttt{C} with type \texttt{T}, the result has type “pointer to member of class \texttt{C} of type \texttt{T}” and is a prvalue designating \texttt{C::m}. Otherwise, if the type of the expression is \texttt{T}, the result has type “pointer to \texttt{T}” and is a prvalue that is the address of the designated object (6.6.1) or a pointer to the designated function. [Note: In particular, the address of an object of type “\texttt{cv T}” is “pointer to \texttt{cv T}”, with the same cv-qualification. —end note] For purposes

\textsuperscript{71} \texttt{const\_cast} is not limited to conversions that cast away a const-qualifier.
The operand of `7
The operand of the unary `1
The operand of prefix `2
The address of an overloaded function (Clause 12) can be taken only in a context that uniquely determines whether the operand is a static or non-static member function, the context can also affect whether the which version of the overloaded function is referred to (see 12.4). 

§ 7.6.2.3 Await [expr.await]

There is an ambiguity in the grammar when `7.6.2.2 Increment and decrement [expr.pre.incr]

The operand of prefix `++ is modified 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. 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 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]

The co_await expression is used to suspend evaluation of a coroutine (9.4.4) while awaiting completion of the computation represented by the operand expression.

© ISO/IEC N4810

10 of pointer arithmetic (7.6.6) and comparison (7.6.9, 7.6.10), an object that is not an array element whose address is taken in this way is considered to belong to an array with one element of type T. [Example:

```c
struct A { int i; };
struct B : A {};
...
&B::i ...
// has type 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 `mutable non-static data member (9.1.1) does not reflect the `mutable specifier associated with the non-static data member. — end note]

4 A pointer to member is only formed when an explicit `& is used and its operand is a `qualified-id not enclosed in parentheses. [Note: 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]

5 If `& is applied to an lvalue of incomplete class type and the complete type declares `operator&(), it is unspecified whether the operator has the built-in meaning or the operator function is called. The operand of `& shall not be a bit-field.

6 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 12.4). [Note: Since the context might determine whether the operand is a static or non-static member function, the context can also affect whether the expression has type “pointer to function” or “pointer to member function”. — end note]

7 The operand of the unary `+ operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument. Integral promotion is performed on integral or enumeration operands. The type of the result is the type of the promoted operand.

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

9 The operand of the logical negation operator `! is contextually converted to `bool (7.3); its value is `true if the converted operand is `false and `false otherwise. The type of the result is `bool.

10 The operand of `~ shall have integral or unscoped enumeration type; the result is the ones’ complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand. There is an ambiguity in the grammar when `~ is followed by a `type-name or `decltype-specifier. The ambiguity is resolved by treating `~ as the unary complement operator rather than as the start of an `unqualified-id naming a destructor. [Note: 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 ambiguously parsed as a destructor name. — end note]

7.6.2.2 Increment and decrement [expr.pre.incr]

1 The operand of prefix `++ is modified 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. 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]

2 The operand of prefix `-- is modified 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]

1 The co_await expression is used to suspend evaluation of a coroutine (9.4.4) while awaiting completion of the computation represented by the operand expression.
An await-expression shall appear only in a potentially-evaluated expression within the compound-statement of a function-body outside of a handler (Clause 14). 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 (9.2.3.6). 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:

1. \( p \) is an lvalue naming the promise object (9.4.4) of the enclosing coroutine and \( P \) is the type of that object.
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 (9.4.4). Otherwise, the unqualified-id await_transform is looked up within the scope of \( P \) by class member access lookup (6.4.5), and if this lookup finds at least one declaration, then \( a \) is \( p\text{.await_transform}(\text{cast-expression}) \); otherwise, \( a \) is the cast-expression.
3. \( o \) is determined by enumerating the applicable operator co_await functions for an argument \( a \) (12.3.1.2), and choosing the best one through overload resolution (12.3). 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.
4. \( e \) is an lvalue referring to the result of evaluating the (possibly-converted) \( o \) expression.
5. \( h \) is an object of type std::coroutine_handle\(<P>\) referring to the enclosing coroutine.
6. \( \text{await-ready} \) is the expression \( e\text{.await_ready()} \), contextually converted to bool.
7. \( \text{await-suspend} \) is the expression \( e\text{.await_suspend(h)} \), which shall be a prvalue of type void, bool, or std::coroutine_handle\(<Z>\) for some type \( Z \).
8. \( \text{await-resume} \) is the expression \( e\text{.await_resume()} \).

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

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

- If the result of await-ready is false, the coroutine is considered suspended. Then:
  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 (9.4.4). — end note]
  2. Otherwise, if the type of await-suspend is bool, await-suspend is evaluated, and the coroutine is resumed if the result is false.
  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 (14.1). Otherwise, control flow returns to the current coroutine caller or resumer (9.4.4) without exiting any scopes (8.6).

- 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.
template <class Rep, class Period>
auto operator co_await(std::chrono::duration<Rep, Period> d) {
    struct awaiter {
        std::chrono::system_clock::duration duration;
        /* ... */
        awaiter(std::chrono::system_clock::duration d) : duration(d) {} 
        bool await_ready() const { return duration.count() <= 0; } 
        void await_resume() {} 
        void await_suspend(std::coroutine_handle<> h) { /* ... */ } 
    }
    return awaiter{d};
}

using namespace std::chrono;

my_future<int> h();

my_future<void> g() {
    std::cout << "just about go to sleep...\n";
    co_await 10ms;
    std::cout << "resumed\n";
    co_await h();
}

auto f(int x = co_await h()); // error: await-expression outside of function suspension context
int a[] = { co_await h() }; // error: await-expression outside of function suspension context

7.6.2.4 Sizeof [expr.sizeof]

1 The 

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 (6.7.1) 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 6.6.1 for the definition of byte and 6.7 for the definition of object representation. — end note]

2 When applied to a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The result of applying sizeof to a potentially-overlapping subobject is the size of the type, not the size of the subobject. When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.

3 The lvalue-to-rvalue (7.3.1), 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 (13.6.3). A sizeof... expression is a pack expansion (13.6.3). [Example: 


5 The result of sizeof and sizeof... is a constant of type std::size_t. [Note: std::size_t is defined in the standard header <cstdlib> (17.2.1, 17.2.4). — end note]
The **new-expression** attempts to create an object of the **type-id** (9.2.1) 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 (6.6.2, 6.7, 11.6.3). [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]

```cpp
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:**

```cpp
(expression-list)
```

**new-type-id:**

```cpp
(type-specifier-seq new-declarator opt)
```

**new-declarator:**

```cpp
ptr-operator new-declarator opt
noptr-new-declarator
```

**noptr-new-declarator:**

```cpp
[ expression opt ] attribute-specifier-seq opt
noptr-new-declarator [ constant-expression ] attribute-specifier-seq opt
```

**new-initializer:**

```cpp
(expression-list opt)
braced-init-list
```

---

1. If a placeholder type (9.1.7.5) 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 \( \text{init} \) 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 (9.1.7.5):

\[
T x \text{init};
\]

[Example:
```cpp
new auto(1); // allocated type is int
class A
```}

2. [Example:
```cpp
template<class T> struct A { A(T, T); }; auto y = new A<int>{1, 2}; // allocated type is A<int>
```]

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

is ill-formed because the binding is

\[
(new \text{int}) ([*10])(); // error
\]

Instead, the explicitly parenthesized version of the **new** operator can be used to create objects of compound types (6.7.2):

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

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

5. Objects created by a **new-expression** have dynamic storage duration (6.6.5.4). [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.

7 Every `constant-expression` in a `noptr-new-declarator` shall be a converted `constant-expression` (7.7) of type `std::size_t` and shall evaluate to a strictly positive value.  

[Example: Given the definition `int n = 42, new float[n][5]` is well-formed (because `n` is the `expression` of a `noptr-new-declarator`), but `new float[5][n]` is ill-formed (because `n` is not a constant expression). — end example]

8 If the `type-id` or `new-type-id` denotes an array type of unknown bound (9.2.3.4), 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` (9.3.1, 9.3.2).

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

10 A `new-expression` may obtain storage for the object by calling an allocation function (6.6.5.4.1). If the `new-expression` terminates by throwing an exception, it may release storage by calling a deallocation function (6.6.5.4.2). 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 shall provide default definitions for the global allocation functions (6.6.5.4, 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 (11.11). 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]

11 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.

12 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`. 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:

13 --- the evaluation of `e1` is sequenced before the evaluation of `e2`, and

14 --- `e2` is evaluated whenever `e1` obtains storage, and

--- end example

74 If the conversion function returns a signed integer type, the second standard conversion converts to the unsigned type `std::size_t` and thus thwarts any attempt to detect a negative value afterwards.
both \( e_1 \) and \( e_2 \) invoke the same replaceable global allocation function, and

if the allocation function invoked by \( e_1 \) and \( e_2 \) is throwing, any exceptions thrown in the evaluation of either \( e_1 \) or \( e_2 \) would be first caught in the same handler, and

the pointer values produced by \( e_1 \) and \( e_2 \) are operands to evaluated \textit{delete-expressions}, and

the evaluation of \( e_2 \) is sequenced before the evaluation of the \textit{delete-expression} whose operand is the pointer value produced by \( e_1 \).

\[\text{Example:}\]

```cpp
def mergeable(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 unmergeable(int x) {
    std::unique_ptr<char[]> a{new char[8]};
    try {
        // Merging this allocation would change its catch handler.
        std::unique_ptr<char[]> b{new char[x]};
        catch (const std::bad_alloc& e) {
            std::cerr << "Allocation failed: " << e.what() << std::endl;
            throw;
        }
    }
}
```

When a \textit{new-expression} calls an allocation function and that allocation has not been extended, the \textit{new-expression} passes the amount of space requested to the allocation function as the first argument of type \texttt{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. For arrays of \texttt{char}, \texttt{unsigned char}, and \texttt{std::byte}, the difference between the result of the \textit{new-expression} and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (6.6.6) 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]

When a \textit{new-expression} calls an allocation function and that allocation has been extended, the size argument to the allocation call shall be no greater than the sum of the sizes for the omitted calls as specified above, plus the size for the extended call had it not been extended, plus any padding necessary to align the allocated objects within the allocated memory.

The \textit{new-placement} syntax is used to supply additional arguments to an allocation function; such an expression is called a \textit{placement new-expression}.

Overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type \texttt{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 \texttt{std::align_val_t}. If the \textit{new-placement} syntax is used, the \textit{initializer-clauses} in its \textit{expression-list} are the succeeding arguments. If no matching function is found and the allocated object type has new-extended alignment, the alignment argument is removed from the argument list, and overload resolution is performed again.

\[\text{Example:}\]

\(\text{new T} \) results in one of the following calls:

\[
\text{operator new}() \ 	ext{(sizeof(T))} \\
\text{operator new}() \ 	ext{(sizeof(T), std::align_val_t(alignof(T)))}
\]

\(\text{new(2,f)} \ T \) 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)

(17.3) — new T[5] results in one of the following calls:
operator new[](sizeof(T) + 5 + x)
operator new[](sizeof(T) + 5 + x, std::align_val_t(alignof(T)))

(17.4) — new(2,f) T[5] results in one of the following calls:
operator new[](sizeof(T) + 5 + x, 2, f)
operator new[](sizeof(T) + 5 + x, std::align_val_t(alignof(T)), 2, f)

Here, each instance of x is a non-negative unspecified value representing array allocation overhead; the result of the new-expression will be offset by this amount from the value returned by operator new[]. This overhead may be applied in all array new-expressions, including those referencing the library function operator new[](std::size_t, void*) and other placement allocation functions. The amount of overhead may vary from one invocation of new to another. — end example

18 [Note: Unless an allocation function has a non-throwing exception specification (14.4), it indicates failure to allocate storage by throwing a std::bad_alloc exception (6.6.5.4.1, 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.

19 [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]

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

(20.1) — If the new-initializer is omitted, the object is default-initialized (9.3). [Note: If no initialization is performed, the object has an indeterminate value. — end note]

(20.2) — Otherwise, the new-initializer is interpreted according to the initialization rules of 9.3 for direct-initialization.

21 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.

22 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 (11.11), and the constructor (11.3.4) selected for the initialization (if any). If the new-expression creates an array of objects of class type, the destructor is potentially invoked (11.3.6).

23 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]

24 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.

25 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 (9.2.3.5), 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 with a parameter of type std::size_t (6.6.5.4.2) 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

75) This may include evaluating a new-initializer and/or calling a constructor.

§ 7.6.2.5
a non-placement allocation function, the normal deallocation function lookup is used to find the matching deallocation function (7.6.2.6) [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; // ill-formed: non-placement deallocation function matches placement allocation function
```
—end example]

26 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.

7.6.2.6 Delete

The `delete-expression` operator destroys a most derived object (6.6.2) or array created by a `new-expression`. The `delete-expression` shall be evaluated exactly once.

```c
delete-expression:
    ::opt delete cast-expression
    ::opt delete [ ] cast-expression
```

The first alternative is a `single-object delete expression`, and the second is an `array delete expression`. Whenever the `delete` keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative. The operand shall be of pointer to object type or of class type. If of class type, the operand is contextually implicitly converted (7.3) to a pointer to object type. The `delete-expression`’s result has type `void`.

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 (6.6.2) representing a base class of such an object (11.6). 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` must match the type of the object allocated by `new`, not the syntax of the `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 destructor or the behavior is undefined. In an array delete expression, if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.

4 The `cast-expression` in a `delete-expression` shall be evaluated exactly once.

5 If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

6 If the value of the operand of the `delete-expression` is not a null pointer value and the selected deallocation function (see below) is not a destroying operator delete, the `delete-expression` will invoke the destructor (if

76) 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.

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

78) For nonzero-length arrays, this is the same as a pointer to the first element of the array created by that `new-expression`. Zero-length arrays do not have a first element.
any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 11.9.2).

7 If the value of the operand of the delete-expression is not a null pointer value, then:

7.1 If the allocation call for the new-expression for the object to be deleted was not omitted and the allocation was not extended (7.6.2.5), the delete-expression shall call a deallocation function (6.6.5.4.2). The value returned from the allocation call of the new-expression shall be passed as the first argument to the deallocation function.

7.2 Otherwise, if the allocation was extended or was provided by extending the allocation of another new-expression, and the delete-expression for every other pointer value produced by a new-expression that had storage provided by the extended new-expression has been evaluated, the delete-expression shall call a deallocation function. The value returned from the allocation call of the extended new-expression shall be passed as the first argument to the deallocation function.

7.3 Otherwise, the delete-expression will not call a deallocation function.

[Note: The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception. —end note] If the value of the operand of the delete-expression is a null pointer value, it is unspecified whether a deallocation function will be called as described above.

8 A program can provide alternative 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 (11.11). —end note]

9 When the keyword delete in a delete-expression is preceded by the unary :: operator, the deallocation function’s name is looked up in global scope. Otherwise, the lookup considers class-specific deallocation functions (11.11). If no class-specific deallocation function is found, the deallocation function’s name is looked up in global scope.

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

10.1 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.

10.2 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.

10.3 If exactly one function remains, that function is selected and the selection process terminates.

10.4 If the deallocation functions have class scope, the one without a parameter of type std::size_t is selected.

10.5 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.

10.6 Otherwise, it is unspecified whether a deallocation function with a parameter of type std::size_t is selected.

11 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 the deleted object suitably adjusted for the array allocation overhead (7.6.2.5) 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, and either the first argument was not the result of a prior call to a
replaceable allocation 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]

12 Access and ambiguity control are done for both the deallocation function and the destructor (11.3.6, 11.11).

7.6.2.7 Alignof [expr.alignof]

1 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 an integral constant of type std::size_t.

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.

7.6.2.8 noexcept operator [expr.unary.noexcept]

1 The noexcept operator determines whether the evaluation of its operand, which is an unevaluated operand
(7.2), can throw an exception (14.1).

   noexcept-expression:
   noexcept ( expression )

2 The result of the noexcept operator is a constant of type bool and is a prvalue.

3 The result of the noexcept operator is true unless the expression is potentially-throwing (14.4).

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.

   cast-expression:
   unary-expression
   ( type-id ) cast-expression

3 Any type conversion not mentioned below and not explicitly defined by the user (11.3.7) 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

5 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 [expr.mprior oper]

1 The pointer-to-member operators \( \rightarrow^* \) and \( .^* \) group left-to-right.

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

2 The binary operator \( .^* \) binds its second operand, which shall be of type “pointer to member of \( T \)” to its first operand, which shall be a glvalue of class \( T \) or of a class of which \( T \) is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

3 The binary operator \( \rightarrow^* \) 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 \( *(E1).^*E2 \).

4 Abbreviating \( pm-expression.^*cast-expression \) as \( E1.^*E2 \), \( E1 \) is called the object expression. If the dynamic type of \( E1 \) does not contain the member to which \( E2 \) refers, the behavior is undefined. Otherwise, the expression \( E1 \) is sequenced before the expression \( E2 \).

5 The restrictions on cv-qualification, and the manner in which the cv-qualifiers of the operands are combined to produce the cv-qualifiers of the result, are the same as the rules for \( E1.\cdot E2 \) 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,

\[
\text{struct S} {  
    S() : i(0) { }  
    \text{mutable int i;}  
};  
\text{void f()}  
\{  
    \text{const S cs;}  
    \text{int S::i = cs::i;} // pm refers to mutable member S::i  
    \text{cs.*pm = 88;} // ill-formed: cs is a const object  
}\n—end note]

6 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: (ptr_to_obj->*ptr_to_mfct)(10); calls the member function denoted by \( \text{ptr_to_mfct} \) for the object pointed to by \( \text{ptr_to_obj} \). —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

§ 7.6.4 126
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 *, /, and % group left-to-right.

```
multiplicative-expression:
  pm-expression
  multiplicative-expression * pm-expression
  multiplicative-expression / pm-expression
  multiplicative-expression % pm-expression
```

2 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.

3 The binary * operator indicates multiplication.

4 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;\[79] if the
quotient a/b is representable in the type of the result, (a/b)*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.

```
additive-expression:
  multiplicative-expression
  additive-expression + multiplicative-expression
  additive-expression - multiplicative-expression
```

For addition, either both operands shall have arithmetic or unscoped enumeration type, or one operand shall
be a pointer to a completely-defined object type and the other shall have integral or unscoped enumeration
type.

2 For subtraction, one of the following shall hold:

(2.1) — both operands have arithmetic or unscoped enumeration type; or
(2.2) — both operands are pointers to cv-qualified or cv-unqualified versions of the same completely-defined
object type; or
(2.3) — the left operand is a pointer to a completely-defined object type and the right operand has integral or
unscoped enumeration type.

3 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.

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.

(4.1) — If P evaluates to a null pointer value and J evaluates to 0, the result is a null pointer value.
(4.2) — Otherwise, if P points to element x[i] of an array object x with n elements, the expressions P + J and
J + P (where J has the value j) point to the (possibly-hypothetical) element x[i + j] if 0 ≤ i + j ≤ n
and the expression P - J points to the (possibly-hypothetical) element x[i - j] if 0 ≤ i - j ≤ n.
(4.3) — Otherwise, the behavior is undefined.

5 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 <cstdlib>
header (17.2).

(5.1) — If P and Q both evaluate to null pointer values, the result is 0.

79) This is often called truncation towards zero.
80) An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1. A
pointer past the last element of an array x of n elements is considered to be equivalent to a pointer to a hypothetical element
x[n] for this purpose; see 6.7.2.
— Otherwise, if \( P \) and \( Q \) point to, respectively, elements \( x[i] \) and \( x[j] \) of the same array object \( x \), the expression \( P - Q \) has the value \( i - j \).

— Otherwise, the behavior is undefined. [Note: If the value \( i - j \) is not in the range of representable values of type \( \text{std::ptrdiff_t} \), the behavior is undefined. — end note]

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

The shift operators << and >> group left-to-right.

\[
\text{shift-expression} \\
\text{additive-expression} \\
\text{shift-expression} \ll \text{additive-expression} \\
\text{shift-expression} \gg \text{additive-expression}
\]

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.

1 The value of \( E1 \ll E2 \) is the unique value congruent to \( E1 \times 2^{E2} \mod 2^N \), 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]

2 The value of \( E1 \gg E2 \) is \( E1/2^{E2} \), 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]

3 The expression \( p \ll q \) is a prvalue indicating whether \( p \) is less than, equal to, greater than, or incomparable with \( q \).

4 If one of the operands is of type \( \text{bool} \) and the other is not, the program is ill-formed.

3 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:

1 — If a narrowing conversion (9.3.4) is required, other than from an integral type to a floating point type, the program is ill-formed.

2 Otherwise, the operands have integral type, the result is of type \( \text{std::strong_ordering} \). The result is \( \text{std::strong_ordering::equal} \) if both operands are arithmetically equal, \( \text{std::strong_ordering::less} \) if the first operand is arithmetically less than the second operand, and \( \text{std::strong_ordering::greater} \) otherwise.

2 — Otherwise, the operands have floating-point type, and the result is of type \( \text{std::partial_ordering} \).

3 The expression \( a \ll b \) yields \( \text{std::partial_ordering::less} \) if \( a \) is less than \( b \), \( \text{std::partial_ordering::greater} \) if \( a \) is greater than \( b \), \( \text{std::partial_ordering::equivalent} \) if \( a \) is equivalent to \( b \), and \( \text{std::partial_ordering::unordered} \) otherwise.

5 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 \( \ll \) to the converted operands.

6 If at least one of the operands is of pointer type, array-to-pointer conversions (7.3.2), 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). If at least one of the operands is of pointer-to-member type, pointer-to-member conversions (7.3.12) and qualification conversions (7.3.5) are performed on both operands to bring them to their composite pointer type (7.2.2). If both operands are null pointer constants, but not both of integer type, pointer conversions (7.3.11) are performed on both operands to bring them to their composite pointer type (7.2.2). In all cases, 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]
7 If the composite pointer type is a function pointer type, a pointer-to-member type, or std::nullptr_t, the result is of type std::strong_equality; the result is std::strong_equality::equal if the (possibly converted) operands compare equal (7.6.10) and std::strong_equality::nonequal if they compare unequal, otherwise the result of the operation is unspecified.
8 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.
9 Otherwise, the program is ill-formed.
10 The five comparison category types (17.11.2) (the types std::strong_ordering, std::strong_equality, std::weak_ordering, std::weak_equality, and std::partial_ordering) are not predefined; if the header <compare> is not 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 (9.1.7.5) in a defaulted three-way comparison (11.10.3) or use of the built-in operator) – the program is ill-formed.

7.6.9 Relational operators [expr.rel]
1 The relational operators group left-to-right. [Example: a<b<c means (a<b)<c and not (a<b)&&b<c). — end example]

relational-expression:
compare-expression
relational-expression < compare-expression
relational-expression > compare-expression
relational-expression <= compare-expression
relational-expression >= compare-expression

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.3).
2 The converted operands shall have arithmetic, enumeration, or pointer type. The operators < (less than), > (greater than), <= (less than or equal to), and => (greater than or equal to) all yield false or true. The type of the result is bool.
3 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.
4 The result of comparing unequal pointers to objects\textsuperscript{81} 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 (11.8), 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.
5 If two operands p and q compare equal (7.6.10), p<=>q and p=>q both yield true and p<q and p>q both yield false. Otherwise, if a pointer p compares greater than a pointer q, p=>q, q<=>p, and q>p all yield true and p<q, p<q, q=>p, and q>p all yield false. Otherwise, the result of each of the operators is unspecified.
6 If both operands (after conversions) are of arithmetic or enumeration type, each of the operators shall yield true if the specified relationship is true and false if it is false.

\textsuperscript{81}An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1. A pointer past the last element of an array x of n elements is considered to be equivalent to a pointer to a hypothetical element x[n] for this purpose; see 6.7.2.
### 7.6.10 Equality operators

*equality-expression:*

`relational-expression`

equality-expression == relational-expression

equality-expression != relational-expression

1. 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.3).

2. The converted operands shall have arithmetic, enumeration, pointer, or pointer-to-member type, or type `std::nullptr_t`. The operators `==` and `!=` both yield `true` or `false`, i.e., a result of type `bool`. In each case below, the operands shall have the same type after the specified conversions have been applied.

3. If at least one of the operands is a pointer, pointer conversions (7.3.11), 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:

   - (3.1) If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object, the result of the comparison is unspecified.
   - (3.2) Otherwise, if the pointers are both null, both point to the same function, or both represent the same address (6.7.2), they compare equal.
   - (3.3) Otherwise, the pointers compare unequal.

4. If at least one of the operands is a pointer to member, pointer-to-member conversions (7.3.12), 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:

   - (4.1) If two pointers to members are both the null member pointer value, they compare equal.
   - (4.2) If only one of two pointers to members is the null member pointer value, they compare unequal.
   - (4.3) If either is a pointer to a virtual member function, the result is unspecified.
   - (4.4) If one refers to a member of class `C1` and the other refers to a member of a different class `C2`, where neither is a base class of the other, the result is unspecified. [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
```
   ]
   - (4.5) If both refer to (possibly different) members of the same union (11.4), they compare equal.
   - (4.6) Otherwise, two pointers to members compare equal if they would refer to the same member of the same most derived object (6.6.2) or the same subobject if indirection with a hypothetical object of the associated class type were performed, otherwise they compare unequal. [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::*pdl)() = pl;
int (D::*pdr)() = pr;
bool x = (pdl == pdr); // false
```

82) An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1.
Two operands of type `std::nullptr_t` or one operand of type `std::nullptr_t` and the other a null pointer constant compare equal.

If two operands compare equal, the result is `true` for the `==` operator and `false` for the `!=` operator. If two operands compare unequal, the result is `false` for the `==` operator and `true` for the `!=` operator. Otherwise, the result of each of the operators is unspecified.

If both operands are of arithmetic or enumeration type, the usual arithmetic conversions (7.4) are performed on both operands; each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

### 7.6.11 Bitwise AND operator

```cpp
bool y = (pb == pl);  // true
```

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 (6.7.1) of the converted operands $x$ and $y$, the coefficient $r_i$ of the base-2 representation of the result $r$ is 1 if both $x_i$ and $y_i$ are 1, and 0 otherwise. [Note: The result is the bitwise AND function of the operands. — end note]

### 7.6.12 Bitwise exclusive OR operator

```cpp
exclusive-or-expression
and-expression ^ exclusive-or-expression
```

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 (6.7.1) of the converted operands $x$ and $y$, the coefficient $r_i$ of the base-2 representation of the result $r$ is 1 if either (but not both) of $x_i$ 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

```cpp
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 (6.7.1) 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

```cpp
logical-and-expression
inclusive-or-expression
```

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`.

The result is a `bool`. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

### 7.6.15 Logical OR operator

```cpp
logical-or-expression
exclusive-or-expression
```

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`.

The result is a `bool`. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second 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, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

### 7.6.16 Conditional operator

conditional-expression:
  logical-or-expression
  logical-or-expression ? expression : assignment-expression

1 Conditional expressions group right-to-left. The first expression is contextually converted to bool (7.3). It is evaluated and if it is true, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated. Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression.

2 If either the second or the third operand has type void, one of the following shall hold:

   (2.1) The second or the third operand (but not both) is a (possibly parenthesized) throw-expression (7.6.18); the result is of the type and value category of the other. The conditional-expression is a bit-field if that operand is a bit-field.

   (2.2) Both the second and the third operands have type void; the result is of type void and is a prvalue.

   [Note: 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 (12.3.3.1) 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:

   (4.1) 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 (9.3.3) to a glvalue.

   (4.2) If E2 is an rvalue, the target type is “rvalue reference to T2”, subject to the constraint that the reference must bind directly.

   (4.3) If E2 is a prvalue or if neither of the conversion sequences above can be formed and at least one of the operands has (possibly cv-qualified) class type:

      (4.3.1) 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,

      (4.3.2) otherwise, if T2 is a base class of T1, the target type is cv1 T2, where cv1 denotes the cv-qualifiers of T1,

      (4.3.3) 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, 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]
If the second and third operands are glvalues of the same value category and have the same type, the result is of that type and value category and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

Otherwise, the result is a prvalue. If the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (12.3.1.2, 12.6). If the overload resolution fails, the program is ill-formed. Otherwise, the conversions thus determined are applied, and the converted operands are used in place of the original operands for the remainder of this subclause.

Lvalue-to-rvalue (7.3.1), 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:

(7.1) The second and third operands have the same type; the result is of that type and the result object is initialized using the selected operand.

(7.2) The second and third operands have arithmetic or enumeration type; the usual arithmetic conversions (7.4) are performed to bring them to a common type, and the result is of that type.

(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 [expr.yield]

`yield-expression`:
  co_yield assignment-expression
  co_yield braced-init-list

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 (9.4.4), then the `yield-expression` is equivalent to the expression `co_await p.yield_value(e)`.

[Example:

```cpp
template <typename T>
struct my_generator {
  struct promise_type {
    T current_value;
    /* ... */
    auto yield_value(T v) {
      current_value = std::move(v);
      return std::suspend_always{};
    }
  }
  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);       // error: yield-expression outside of function suspension context
int a[] = { co_yield 1 };         // error: yield-expression outside of function suspension context
```]
```cpp
int main() {
    auto r1 = g1();
    auto r2 = g2();
    assert(std::equal(r1.begin(), r1.end(), r2.begin(), r2.end()));
}
```

### 7.6.18 Throwing an exception

**throw-expression**

- `throw assignment-expressionopt`

A *throw-expression* is of type `void`.

1. Evaluating a *throw-expression* with an operand throws an exception (14.1); 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`”.

2. A *throw-expression* with no operand rethrows the currently handled exception (14.3). 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:

   ```cpp
   try {
     // ...
   } catch (...) {
     // catch all exceptions
     // respond (partially) to exception
     throw;
     // pass the exception to some other handler
   }
   ```

4. If no exception is presently being handled, evaluating a *throw-expression* with no operand calls `std::terminate()` (14.5.1).

### 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 shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. *— end note*

1. The assignment operator (`=`) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand; their result is an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. The right operand is sequenced before the left operand. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. *Note:* Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. *— end note*

2. In simple assignment (`=`), the object referred to by the left operand is modified by replacing its value with the result of the right operand.

3. If the left operand is not of class type, the expression is implicitly converted (7.3) to the cv-unqualified type of the left operand.

4. If the left operand is of class type, the class shall be complete. Assignment to objects of a class is defined by the copy/move assignment operator (11.3.5, 12.5.3).

5. *Note:* For class objects, assignment is not in general the same as initialization (9.3, 11.3.4.2, 11.3.5, 11.9). *— end note*

6. 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.
7 The behavior of an expression of the form \( E_1 \ op = E_2 \) is equivalent to \( E_1 = E_1 \ op E_2 \) except that \( E_1 \) is evaluated only once. In \( += \) 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.

8 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]

9 A braced-init-list may appear on the right-hand side of

(9.1) — 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 \( x = T\{\} \).

(9.2) — 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 (12.5.3, 12.3).

[Example:

```
complex<double> z;
z = { 1,2 }; // meaning z.operator=({1,2})
z += { 1, 2 }; // meaning z.operator+=({1,2})
int a, b;
a = b = { 1 }; // meaning a=b=1;
a = { 1 } = b; // syntax error
— end example]

7.6.20 Comma operator [expr.comma]

1 The comma operator groups left-to-right.

expression:
  assignment-expression
expression, assignment-expression

A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression (7.2). Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression. 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. If the right operand is a temporary expression (6.6.7), the result is a temporary expression.

2 In contexts where comma is given a special meaning, [Example: in lists of arguments to functions (7.6.1.2) and lists of initializers (9.3) — end example] the comma operator as described in this subclause can appear only in parentheses. [Example:

```
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value 5. — end example]

7.7 Constant expressions [expr.const]

1 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 (11.9.5) is not performed, are called constant expressions. [Note: Constant expressions can be evaluated during translation. — end note]

constant-expression:
  conditional-expression

2 A constant initializer for a variable or temporary object \( o \) is an initializer for which interpreting its full-expression as a constant-expression results in a constant expression, except that if \( o \) is an object, such an initializer 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]

§ 7.7
A variable is *usable in constant expressions* after its initializing declaration is encountered if it is a constexpr variable, or it is of reference type or of const-qualified integral or enumeration type, and its initializer is a constant initializer.

4 An expression \(e\) is a *core constant expression* unless the evaluation of \(e\), following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:

\[(4.1)\]
- **this** (7.5.2), except in a constexpr function or a constexpr constructor that is being evaluated as part of \(e\);

\[(4.2)\]
- an invocation of a function other than a constexpr constructor for a literal class, a constexpr function, or an implicit invocation of a trivial destructor (11.3.6) [Note: Overload resolution (12.3) is applied as usual — end note] ;

\[(4.3)\]
- an invocation of an undefined constexpr function or an undefined constexpr constructor;

\[(4.4)\]
- an invocation of an instantiated constexpr function or constexpr constructor that fails to satisfy the requirements for a constexpr function or constexpr constructor (9.1.5);

\[(4.5)\]
- an expression that would exceed the implementation-defined limits (see Annex B);

\[(4.6)\]
- 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] ;

\[(4.7)\]
- an lvalue-to-rvalue conversion (7.3.1) unless it is applied to

\[(4.7.1)\]
- a non-volatile glvalue that refers to an object that is usable in constant expressions, or

\[(4.7.2)\]
- a non-volatile glvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of \(e\);

\[(4.8)\]
- 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;

\[(4.9)\]
- 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 \(e\);

\[(4.10)\]
- an *id-expression* that refers to a variable or data member of reference type unless the reference has a preceding initialization and either

\[(4.10.1)\]
- it is usable in constant expressions or

\[(4.10.2)\]
- its lifetime began within the evaluation of \(e\);

\[(4.11)\]
- a checked contract (9.11.4) whose predicate evaluates to **false**;

\[(4.12)\]
- 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 (6.2, 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 = *k;   // ill-formed, k 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 have captures to automatic objects created during constant expression evaluation.

```c
static_assert(bind(monad(2))(monad)() == monad(2)());
```

— end example] — end note]

\[(4.13)\]
- a conversion from type \(cv\) **void** to a pointer-to-object type;
— a reinterpret_cast (7.6.1.9);
(4.15) — a pseudo-destructor call (7.5.4.3);
(4.16) — 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;
(4.17) — a new-expression (7.6.2.5);
(4.18) — a delete-expression (7.6.2.6);
(4.19) — an await-expression (7.6.2.3);
(4.20) — a yield-expression (7.6.17);
(4.21) — a three-way comparison (7.6.8), relational (7.6.9), or equality (7.6.10) operator where the result is
unspecified;
(4.22) — 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; or
(4.23) — 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.

Examples:

```c
int x;  // not constant
struct A {
  constexpr A(bool b) : m(b?42:x) { }
  int m;
};
constexpr int v = A(true).m;  // OK: constructor call initializes m with the value 42
constexpr int w = A(false).m;  // error: initializer for m is x, which is non-constant
constexpr int f1(int k) {
  constexpr int x = k;  // error: x is not initialized by a constant expression
  // because lifetime of k began outside the expression incr(k)
  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)
```

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 (11.3.9), as enumerator initializers if the underlying type is not fixed (9.6), and as alignments (9.11.2). — end note]

6 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:

```cpp
struct A {
    constexpr A(int i) : val(i) {}
    constexpr operator int() const { return val; }
    constexpr operator long() const { return 42; }
private:
    int val;
};
template<int> struct X {
};
constexpr A a = alignof(int);
alignas(a) int n; // error: ambiguous conversion
struct B { int n : a; }; // error: ambiguous conversion
```
— end example]

7 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

- user-defined conversions,
- lvalue-to-rvalue conversions (7.3.1),
- array-to-pointer conversions (7.3.2),
- function-to-pointer conversions (7.3.3),
- qualification conversions (7.3.5),
- integral promotions (7.3.6),
- integral conversions (7.3.8) other than narrowing conversions (9.3.4),
- null pointer conversions (7.3.11) from std::nullptr_t,
- null member pointer conversions (7.3.12) from std::nullptr_t, and
- function pointer conversions (7.3.13),

and where the reference binding (if any) binds directly. [Note: Such expressions may be used in new expressions (7.6.2.5), as case expressions (8.4.2), as enumerator initializers if the underlying type is fixed (9.6), as array bounds (9.2.3.4), and as non-type template arguments (13.3). — 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.

8 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:

```cpp
consteval int f() { return 42; }
consteval auto g() { return 7; }
consteval int h(int (*p)() = g()) { return p(); }
constexpr int r = h();   // OK
constexpr auto e = g();  // ill-formed: a pointer to an immediate function is
```

§ 7.7
// 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:

```cpp
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 \texttt{true} or \texttt{false}. — end example] — end note]

An expression or conversion is in an \textit{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 \textit{immediate invocation} if it is an explicit or implicit invocation of an immediate function and is not in an immediate function context. An immediate invocation shall be a constant expression. [Note: An immediate invocation is evaluated even in an unevaluated operand. — end note]

An expression or conversion \( e \) is \textit{manifestly constant-evaluated} if it is:

(11.1) — a \textit{constant-expression}, or
(11.2) — the condition of a \textit{constexpr if} statement (8.4.1), or
(11.3) — an immediate invocation, or
(11.4) — a \textit{constraint-expression} (13.4.2) including one formed from the \textit{constraint-logical-or-expression} of a \textit{requires-clause}, or
(11.5) — the initializer of a variable that is usable in constant expressions or has constant initialization.

[Example:

```cpp
template<bool> struct X {}; 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]; // ill-formed: a is not usable
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);
}
```

— end example]

An expression or conversion is \textit{potentially constant evaluated} if it is:

(12.1) — a manifestly constant-evaluated expression,
(12.2) — a potentially-evaluated expression (6.2),
(12.3) — an immediate subexpression of a \textit{braced-init-list},
(12.4) — an expression of the form \& \textit{cast-expression} that occurs within a templated entity, or
(12.5) — a subexpression of one of the above that is not a subexpression of a nested unevaluated operand.

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.
Constant evaluation may be necessary to determine whether a narrowing conversion is performed (9.3.4).
Constant evaluation may be necessary to determine whether such an expression is value-dependent (13.7.2.3).
A function or variable is needed for constant evaluation if it is:

(12.6) a constexpr function that is named by an expression (6.2) that is potentially constant evaluated, or

(12.7) 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.

[Example:

```cpp
struct N {
    constexpr N() {}
    N(N const&) = delete;
};
template<typename T> constexpr void bad_assert_copyable() { T t; T t2 = t; }
using ineffective = decltype(bad_assert_copyable<N>());
// bad_assert_copyable<N> is not needed for constant evaluation
// (and thus not instantiated)
template<typename T> constexpr void assert_copyable() { T t; T t2 = t; }
using check = decltype(assert_copyable<N>());
// error: assert_copyable<N> is instantiated (because it is needed for constant evaluation), but the attempt to copy t is ill-formed

— end example]
## 8 Statements

1. Except as indicated, statements are executed in sequence.

   - **statement:**
     - labeled-statement
     - attribute-specifier-seq\textsubscript{opt}, expression-statement
     - attribute-specifier-seq\textsubscript{opt}, compound-statement
     - attribute-specifier-seq\textsubscript{opt}, selection-statement
     - attribute-specifier-seq\textsubscript{opt}, iteration-statement
     - attribute-specifier-seq\textsubscript{opt}, jump-statement
     - declaration-statement
     - attribute-specifier-seq\textsubscript{opt}, try-block

   - **init-statement:**
     - expression-statement
     - simple-declaration

   - **condition:**
     - expression
     - attribute-specifier-seq\textsubscript{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 $S_1$ encloses a statement $S_2$ if
   - (3.1) $S_2$ is a substatement of $S_1$ (Clause 9),
   - (3.2) $S_1$ is a selection-statement or iteration-statement and $S_2$ is the init-statement of $S_1$,
   - (3.3) $S_1$ is a try-block and $S_2$ is its compound-statement or any of the compound-statements of its handlers, or
   - (3.4) $S_1$ encloses a statement $S_3$ and $S_3$ encloses $S_2$.

4. The rules for conditions apply both to selection-statements and to the for and while statements (8.5). 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 9.1.7.5.

5. A name introduced by a declaration in a condition (introduced by either the decl-specifier-seq or the declarator of the condition) is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is redeclared in the outermost block of a substatement controlled by the condition, the declaration that redeclares the name is ill-formed. [Example:

   ```c
   if (int x = f()) {
     int x; // ill-formed, redeclaration of x
   } else {
     int x; // ill-formed, redeclaration of x
   }
   —end example]

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

---

**Statements**

141
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.

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.

In the decl-specifier-seq of a condition, each declSpecifier shall be either a typeSpecifier or constexpr.

8.1 Labeled statement

A statement can be labeled.

labeled-statement:
  attribute-specifier-seqopt identifier : statement
  attribute-specifier-seqopt case constant-expression : statement
  attribute-specifier-seqopt 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 (6.4.1) ignores labels. — end note]

Case labels and default labels shall occur only in switch statements.

8.2 Expression statement

Expression statements have the form

expression-statement:
  expressionopt ;

The expression is a discarded-value expression (7.2). 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 (8.5.1). — end note]

8.3 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.

compound-statement:
  { statement-seqopt }

A compound statement defines a block scope (6.3). [Note: A declaration is a statement (8.7). — end note]

8.4 Selection statements

Selection statements choose one of several flows of control.

selection-statement:
  if constexpropt ( init-statementseqopt condition ) statement
  if constexpropt ( init-statementseqopt condition ) statement else statement
  switch ( init-statementseqopt condition ) statement

See 9.2.3 for the optional attribute-specifier-seq in a condition. [Note: An init-statement ends with a semicolon. — end note]

The substatement in a selection-statement (each substatement, in the else form of the if statement) implicitly defines a block scope (6.3). 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:
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]  

8.4.1 The if statement

If the condition (8.4) 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.  

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 (Clause 13), if the condition is not value-dependent after its instantiation, the discarded substatement (if any) is not instantiated. [Note: Odr-uses (6.2) 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 (8.4.2) within the same if statement. A label (8.1) declared in a substatement of a constexpr if statement shall only be referred to by a statement (8.6.4) in the same substatement. [Example:

```cpp
template<typename T, typename ... Rest> void g(T&& p, Rest&& ... rs) {
    // ... handle p
    if constexpr (sizeof...(rs) > 0)
        g(rs...); // never instantiated with an empty argument list
}
```

extern int x; // no definition of x required

int f() {
    if constexpr (true)
        return 0;
    else if (x)
        return x;
    else
        return -x;
}

—end example]

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
}

87) In other words, the else is associated with the nearest un-elsed if.  

§ 8.4.1
8.4.2 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, 8.6.1. [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.

8.5 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 decl-specifier-seq declarator
    attribute-specifier-seq_opt decl-specifier-seq ref-qualifier_opt [ identifier-list ]

for-range-initializer:
    expr-or-braced-init-list

See 9.2.3 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 (6.3) 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:

```c
while (--x >= 0)
  int i;
```
can be equivalently rewritten as

```c
while (--x >= 0) {
  int i;
}
```
Thus after the while statement, i is no longer in scope. — end example]

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]

8.5.1 The while statement [stmt.while]
1 In the while statement the substatement is executed repeatedly until the value of the condition (8.4) 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 (6.3.2) 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]

8.5.2 The do statement [stmt.do]
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.

8.5.3 The for statement [stmt.for]
1 The for statement

```c
for ( init-statement condition_opt ; expression_opt ) statement
```
is equivalent to

```
{ 
  init-statement
  while ( condition ) {
    statement
    expression ;
  }
}
```

except that names declared in the `init-statement` are in the same declarative region as those declared in the `condition`, and except that a `continue` in `statement` (not enclosed in another iteration statement) will execute `expression` before re-evaluating `condition`. [Note: Thus the first statement specifies initialization for the loop; the condition (8.4) specifies a test, sequenced before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is sequenced after each iteration. — end note]

2 Either or both of the `condition` and the `expression` can be omitted. A missing `condition` makes the implied `while` clause equivalent to `while(true)`.

3 If the `init-statement` is a declaration, the scope of the name(s) declared extends to the end of the `for` statement. [Example:

```c
int i = 42;
int a[10];
for (int i = 0; i < 10; i++)
a[i] = i;
int j = i;  // j = 42
```

— end example]

§ 8.5.4  The range-based for statement  [stmt.ranged]  

1 The range-based for statement

```
for ( init-statement,opt for-range-declaration : for-range-initializer ) statement
```

is equivalent to

```
{ 
  init-statement,opt
  auto &range = for-range-initializer ;
  auto begin = begin-expr ;
  auto end = end-expr ;
  for ( ; begin != end ; ++begin ) {
    for-range-declaration = *begin ;
    statement
  }
}
```

where

(1.1) — if the `for-range-initializer` is an `expression`, it is regarded as if it were surrounded by parentheses (so that a comma operator cannot be reinterpreted as delimiting two `init-declarators`);

(1.2) — `range`, `begin`, and `end` are variables defined for exposition only; and

(1.3) — `begin-expr` and `end-expr` are determined as follows:

(1.3.1) — if the `for-range-initializer` is an expression of array type R, `begin-expr` and `end-expr` are `range` and `range + N`, respectively, where N is the array bound. If R is an array of unknown bound or an array of incomplete type, the program is ill-formed;

(1.3.2) — if the `for-range-initializer` is an expression of class type C, the `unqualified-ids` `begin` and `end` are looked up in the scope of C as if by class member access lookup (6.4.5), and if both find at least one declaration, `begin-expr` and `end-expr` are `range.begin()` and `range.end()`, respectively;

(1.3.3) — otherwise, `begin-expr` and `end-expr` are `begin(range)` and `end(range)`, respectively, where `begin` and `end` are looked up in the associated namespaces (6.4.2). [Note: Ordinary unqualified lookup (6.4.1) is not performed. — end note]
Example:

```cpp
typearray{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.

8.6 Jump statements

Jump statements unconditionally transfer control.

```cpp
class jump-statement:
  break;
  continue;
  return expr-or-braced-init-list_opt;
  coroutine-return-statement
  goto identifier;
```

On exit from a scope (however accomplished), objects with automatic storage duration (6.6.5.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [Note: For temporaries, see 6.6.7. — 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 8.7 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]

8.6.1 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.

8.6.2 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

```cpp
while (foo) {
  // ...
  // ...
  // ...
  contin:;
  contin:;
  contin:;
} while (foo);
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

8.6.3 The `return` statement

A function returns to its caller by the `return` statement.

The `expr-or-braced-init-list` of a `return` statement is called its operand. A `return` statement with no operand shall be used only in a function whose return type is `cv void`, a constructor (11.3.4), or a destructor (11.3.6). 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 gvalue result or prvalue result object of the (explicit or implicit) function call by copy-initialization (9.3) 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 (11.9.5). — end note] [Example:

```cpp
std::pair<std::string, int> f(const char* p, int x) {
  return {p,x};
}
```
Flowing off the end of a constructor, a destructor, or a non-coroutine function with a \texttt{cv void} return type is equivalent to a \texttt{return} with no operand. Otherwise, flowing off the end of a function other than \texttt{main (6.8.3.1)} or a coroutine (9.4.4) results in undefined behavior.

3 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 \texttt{return} statement, which, in turn, is sequenced before the destruction of local variables (8.6) of the block enclosing the \texttt{return} statement.

8.6.3.1 The \texttt{co\_return} statement

\texttt{coroutine-return-statement:}
\begin{verbatim}
co_return expr-or-braced-init-list_opt ;
\end{verbatim}

1 A coroutine returns to its caller or resumer (9.4.4) by the \texttt{co\_return} statement or when suspended (7.6.2.3). A coroutine shall not return to its caller or resumer by a \texttt{return} statement (8.6.3).

2 The \texttt{expr-or-braced-init-list} of a \texttt{co\_return} statement is called its operand. Let \textit{p} be an lvalue naming the coroutine promise object (9.4.4). A \texttt{co\_return} statement is equivalent to:

\begin{verbatim}
{ S; goto final-suspend; }
\end{verbatim}

where \textit{final-suspend} is the exposition-only label defined in 9.4.4 and \textit{S} is defined as follows:

(2.1) \textit{S} is \texttt{p\_return\_value(expr-or-braced-init-list)}, if the operand is a \texttt{braced-init-list} or an expression of non-void type;

(2.2) \textit{S} is \{ \texttt{expression_opt; p\_return\_void(); \}}, otherwise;

\textit{S} shall be a prvalue of type \texttt{void}.

3 If \texttt{p\_return\_void()} is a valid expression, flowing off the end of a coroutine is equivalent to a \texttt{co\_return} with no operand; otherwise flowing off the end of a coroutine results in undefined behavior.

8.6.4 The \texttt{goto} statement

1 The \texttt{goto} statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (8.1) located in the current function.

8.7 Declaration statement

1 A declaration statement introduces one or more new identifiers into a block; it has the form

\begin{verbatim}
declaration-statement:
  block-declaration
\end{verbatim}

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.

2 Variables with automatic storage duration (6.6.5.3) are initialized each time their \textit{declaration-statement} is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (8.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization (including ones in \textit{conditions} and \textit{init-statement}s). A program that jumps\textsuperscript{88} 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 (9.3). In such a case, the variables with vacuous initialization are constructed in the order of their declaration. [Example:

\begin{verbatim}
void f() { // ...
goto lx; // ill-formed: jump into scope of a
// ...
ly: X a = 1; // ...
lx: goto ly; // OK, jump implies destructor call for a followed by
// construction again immediately following label ly
}
\end{verbatim}

\textsuperscript{88}) The transfer from the condition of a \texttt{switch} statement to a \texttt{case} label is considered a jump in this respect.
Dynamic initialization of a block-scope variable with static storage duration (6.6.5.1) or thread storage duration (6.6.5.2) 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. 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);  // recursive call - undefined
    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: 6.8.3.4 describes the order in which block-scope objects with static and thread storage duration are destroyed. —end note]

### 8.8 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 (9.1.7),

```c
T(a)->m = 7;          // expression-statement
T(a)++;               // expression-statement
T(a,5)<<c;            // expression-statement
T(*d)(int);           // declaration
T(e)[5];              // declaration
T(f) = { 1, 2 };      // declaration
T(*g)(double(3));     // declaration
```

In the last example above, g, which is a pointer to T, is initialized to double(3). This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. —end example]

The remaining cases are declarations. [Example:

```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

---

89) The implementation must not introduce any deadlock around execution of the initializer. Deadlocks might still be caused by the program logic; the implementation need only avoid deadlocks due to its own synchronization operations.
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),
    (**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 [dcl.dcl]

1 Declarations generally specify how names are to be interpreted. Declarations have the form

\[\text{declaration-seq}:\]
\[\text{declaration}\]
\[\text{declaration-seq} \text{ declaration}\]

\[\text{declaration}:\]
\[\text{block-declaration}\]
\[\text{nodeclspec-function-declaration}\]
\[\text{function-definition}\]
\[\text{template-declaration}\]
\[\text{declaration-guide}\]
\[\text{explicit-instantiation}\]
\[\text{explicit-specialization}\]
\[\text{export-declaration}\]
\[\text{linkage-specification}\]
\[\text{namespace-specification}\]
\[\text{empty-declaration}\]
\[\text{attribute-declaration}\]

\[\text{block-declaration}:\]
\[\text{simple-declaration}\]
\[\text{asm-definition}\]
\[\text{namespace-alias-definition}\]
\[\text{using-declaration}\]
\[\text{using-directive}\]
\[\text{static_assert-declaration}\]
\[\text{alias-declaration}\]
\[\text{opaque-enum-declaration}\]

\[\text{nodeclspec-function-declaration}:\]
\[\text{attribute-specifier-seq}_{\text{opt}} \text{ declarator} ;\]

\[\text{alias-declaration}:\]
\[\text{using} \text{ identifier} \text{ attribute-specifier-seq}_{\text{opt}} = \text{defining-type-id} ;\]

\[\text{simple-declaration}:\]
\[\text{decl-specifier-seq} \text{ init-declarator-list}_{\text{opt}} ;\]
\[\text{attribute-specifier-seq} \text{ decl-specifier-seq} \text{ init-declarator-list} ;\]
\[\text{attribute-specifier-seq}_{\text{opt}} \text{ decl-specifier-seq} \text{ ref-qualifier}_{\text{opt}} [ \text{identifier-list} ] \text{ initializer} ;\]

\[\text{static_assert-declaration}:\]
\[\text{static_assert} ( \text{constant-expression} ) ;\]
\[\text{static_assert} ( \text{constant-expression} , \text{string-literal} ) ;\]

\[\text{empty-declaration}:\]
\[;\]

\[\text{attribute-declaration}:\]
\[\text{attribute-specifier-seq} ;\]

[Note: \text{asm-defined} functions are described in 9.9, and \text{linkage-specifications} are described in 9.10; \text{function-defined} functions are described in 9.4 and \text{template-declarations} and \text{deduction-guides} are described in Clause 13; \text{namespace-defined} functions are described in 9.7.1, using-declarations are described in 9.8 and using-directives are described in 9.7.3. —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 9.11. \text{decl-specifiers}, the principal components of a \text{decl-specifier-seq}, are described in 9.1. \text{declarators}, the components of an \text{init-declarator-list}, are described in 9.2. The \text{attribute-specifier-seq} appertains to each of the entities declared by the \text{declarators} of the \text{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 \text{declarator-id} for that declaration. —end note] [Example:
Except where otherwise specified, the meaning of an attribute-declaration is implementation-defined.

A declaration occurs in a scope (6.3); the scope rules are summarized in 6.4. A declaration that declares a function or defines a class, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in Clause 9 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are not nested within scopes nested within the declaration.

In a simple-declaration, the optional init-declarator-list can be omitted only when declaring a class (Clause 11) or enumeration (9.6), that is, when the decl-specifier-seq contains either a class-specifier, an elaborated-type-specifier with a class-key (11.2), or an enum-specifier. In these cases and whenever a class-specifier or enum-specifier is present in the decl-specifier-seq, the identifiers in these specifiers are among the names being declared by the declaration (as class-names, enum-names, or enumerators, depending on the syntax). In such cases, the decl-specifier-seq shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration. [Example:

```c
enum { }; // ill-formed
typedef class { }; // ill-formed
```
—end example]

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]

An empty-declaration has no effect.

A simple-declaration with an identifier-list is called a structured binding declaration (9.5). If the decl-specifier-seq contains any decl-specifier other than static, thread_local, auto (9.1.7.5), 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.

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 (9.1.7) in the decl-specifier-seq and the recursive declarator structure of the init-declarator describe a type (9.2.3), which is then associated with the name being declared by the init-declarator.

If the decl-specifier-seq contains the typedef specifier, the declaration is called a typedef declaration and the name of each init-declarator is declared to be a typedef-name, synonymous with its associated type (9.1.3). 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 (9.2.3.5) and an object declaration otherwise.

Syntactic components beyond those found in the general form of declaration are added to a function declaration to make a function-definition. An object declaration, however, is also a definition unless it contains the extern specifier and has no initializer (6.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (9.3) to be done.

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 (Clause 13), explicit-instantiation (13.8.2), or explicit-specialization (13.8.3). —end note]

### 9.1 Specifiers

The specifiers that can be used in a declaration are

```c
[[noreturn]] void f [[noreturn]] (); // OK
```
—end example]
The optional attribute-specifier-seq in a decl-specifier-seq appertains to the type determined by the preceding decl-specifiers (9.2.3). The attribute-specifier-seq affects the type only for the declaration it appears in, not other declarations involving the same type.

2 Each decl-specifier shall appear at most once in a complete decl-specifier-seq, except that long may appear twice. The constexpr and consteval decl-specifiers shall not both appear in a decl-specifier-seq.

3 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:

typedef char* 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,

void f(const Pc); // void f(char* const) (not const char*)
void g(const int Pc); // void g(const int)
—end example]

4 [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:

void h(unsigned Pc); // void h(unsigned int)
void k(unsigned int Pc); // void k(unsigned int)
—end example] — end note]

9.1.1 Storage class specifiers [dcl.stc]

The storage class specifiers are

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 (11.4.1)). 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. A storage-class-specifier other than thread_local shall not be specified in an explicit specialization (13.8.3) or an explicit instantiation (13.8.2) directive.

2 [Note: A variable declared without a storage-class-specifier at block scope or declared as a function parameter has automatic storage duration by default (6.6.5.3). — end note]

3 The thread_local specifier indicates that the named entity has thread storage duration (6.6.5.2). It shall be applied only to the declaration of a variable of namespace or block scope, to a structured binding
declaration (9.5), 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 (9.5), or to the declaration of an anonymous union (11.4.1). 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 (6.6.5.1), unless accompanied by the `thread_local` specifier, which declares the variable to have thread storage duration (6.6.5.2). A `static` specifier can be used in declarations of class members; 11.3.8 describes its effect. For the linkage of a name declared with a `static` specifier, see 6.5.

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 6.5. [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. Each function in a given set of overloaded functions can have a different linkage, however. [Example:

```c
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(); void l(); // external linkage
inline void m(); extern void m(); // external linkage
static void n(); inline void n(); // internal linkage

static int a; // a has internal linkage
int a; // error: two definitions
static int b; // b has internal linkage
extern int b; // b still has internal linkage
int c; // c has external linkage
static int c; // error: inconsistent linkage
extern int d; // d has external linkage
static int d; // error: inconsistent linkage
```
—end example]

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:

```c
struct S;
extern S a;
extern S f();
extern void g(S);

void h() {
    g(a); // error: S is incomplete
```
§ 9.1.2 Function specifiers

A function-specifier can be used only in a function declaration.

\[
\text{function-specifier:} \\
\quad \text{virtual} \\
\quad \text{explicit-specifier}
\]

\[
\text{explicit-specifier:} \\
\quad \text{explicit ( constant-expression )} \\
\quad \text{explicit}
\]

1. The virtual specifier shall be used only in the initial declaration of a non-static class member function; see 11.6.2.

2. An explicit-specifier shall be used only in the declaration of a constructor or conversion function within its class definition; see 11.3.7.1 and 11.3.7.2.

3. In an explicit-specifier, the constant-expression, if supplied, shall be a contextually converted constant expression of type bool (7.7). The explicit-specifier explicit without a constant-expression is equivalent to the explicit-specifier explicit(true). If the constant expression evaluates to true, the function is explicit. Otherwise, the function is not explicit. A ( token that follows explicit is parsed as part of the explicit-specifier.

9.1.3 The typedef specifier

Declarations containing the decl-specifier typedef declare identifiers that can be used later for naming fundamental (6.7.1) or compound (6.7.2) types. The typedef specifier shall not be combined in a decl-specifier-seq with any other kind of specifier except a defining-type-specifier, and it shall not be used in the decl-specifier-seq of a parameter-declaration (9.2.3.5) nor in the decl-specifier-seq of a function-definition (9.4).

If a typedef specifier appears in a declaration without a declarator, the program is ill-formed.

\[
\text{typedef-name:} \\
\quad \text{identifier} \\
\quad \text{simple-template-id}
\]

A name declared with the typedef specifier becomes a typedef-name. A typedef-name names the type associated with the identifier (9.2) or simple-template-id (Clause 13); a typedef-name is thus a synonym for another type. A typedef-name does not introduce a new type the way a class declaration (11.2) or enum declaration (9.6) does. [Example: After]

    typedef int MILES, *KLICKSP;

the constructions

    MILES distance;
    extern KLICKSP metricp;

are all correct declarations; the type of distance is int and that of metricp is “pointer to int.” — end example]

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:
using handler_t = void (*)(int);
extern handler_t ignore;
extern void (*ignore)(int);   // redeclare ignore
using cell = pair<void*, cell*>;   // ill-formed
— end example] The defining-type-specifier-seq of the defining-type-id shall not define a class or enumeration
if the alias-declaration is the declaration of a template-declaration.

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:

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:

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:

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:

class complex { /* ... */ };  
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:

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 (11.2). If a typedef-name is used to identify the subject of an elaborated-type-specifier (9.1.7.3), a class
definition (Clause 11), a constructor declaration (11.3.4), or a destructor declaration (11.3.6), the program is
ill-formed. — end note] [Example:

struct S {
    S();
    ~S();
};
typedef struct S T;
S a = T();   // OK
struct T* p;   // error
9 If the typedef declaration defines an unnamed class (or enum), the first typedef-name declared by the declaration to be that class type (or enum type) is used to denote the class type (or enum type) for linkage purposes only (6.5). [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:
  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]

9.1.4 The friend specifier

The friend specifier is used to specify access to class members; see 11.8.3.

9.1.5 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 constexpr 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 (9.1.6). 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:
  constexpr void square(int &x);  // OK: declaration
  constexpr int bufsz = 1024;       // OK: definition
  constexpr struct pixel {
    int x;
    int y;
    constexpr pixel(int);  // OK: declaration
  };
  constexpr pixel::pixel(int a) : x(a), y(x)  // OK: definition
  { square(x); }
  constexpr pixel small(2);  // error: square not defined, so small(2)
  // not constant (7.7) so constexpr not satisfied

  constexpr void square(int &x) { // OK: definition
    x *= x;
  }
  constexpr pixel large(4);  // OK: square defined
  int next(constexpr int x) { // error: not for parameters
    return x + 1;
  }
  extern constexpr int memsz;  // error: not a definition
— end example]

A constexpr or consteval specifier used in the declaration of a function that is not a constructor declares that function to be a constexpr function. Similarly, a constexpr or consteval specifier used in a constructor declaration declares that constructor to be a constexpr constructor. A function or constructor declared with the constexpr specifier is called an immediate function. A destructor, an allocation function, or a deallocation function shall not be declared with the constexpr specifier.

The definition of a constexpr function shall satisfy the following requirements:

(3.1) — its return type shall be a literal type;
(3.2) — each of its parameter types shall be a literal type;
(3.3) — it shall not be a coroutine (9.4.4);
(3.4) — its function-body shall not enclose (Clause 8)
(3.4.1) — an asm-definition,
(3.4.2) — a goto statement,
— an identifier label (8.1),
— a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed.

[Note: A function-body that is = delete or = default encloses none of the above. — end note]

[Example:
```cpp
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 uninit() {
  int a;
  // error: variable is uninitialized
  return 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 shall satisfy the following requirements:

(4.1) — the class shall not have any virtual base classes;
(4.2) — each of the parameter types shall be a literal type.

In addition, either its function-body shall be = delete, or it shall satisfy the following requirements:

(4.3) — either its function-body shall be = default, or the compound-statement of its function-body shall satisfy the requirements for a function-body of a constexpr function;
(4.4) — every non-variant non-static data member and base class subobject shall be initialized (11.9.2);
(4.5) — if the class is a union having variant members (11.4), exactly one of them shall be initialized;
(4.6) — if the class is a union-like class, but is not a union, for each of its anonymous union members having variant members, exactly one of them shall be initialized;
(4.7) — for a non-delegating constructor, every constructor selected to initialize non-static data members and base class subobjects shall be a constexpr constructor;
(4.8) — for a delegating constructor, the target constructor shall be a constexpr constructor.

[Example:
```cpp
struct Length {
  constexpr explicit Length(int i = 0) : val(i) { }
private:
  int val;
};
— end example]

5 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, a constant initializer for some object (6.8.3.2), 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]
friend function declaration, that declaration shall be a definition or the function shall have previously been declared inline.

6 If an inline function or variable is odr-used in a translation unit, a definition of it shall be reachable from the end of that translation unit, and it shall have exactly the same definition in every such translation unit (6.2). [Note: A call to the inline function or a use of the inline variable may be encountered before its definition appears in the translation unit. — end note] 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 translation unit, there shall be a reachable inline declaration in all translation units in which it is declared; no diagnostic is required. An inline function or variable with external or module linkage shall have the same address in all translation units. [Note: 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]

7 An exported inline function or variable shall be defined in the translation unit containing its exported declaration, outside the private-module-fragment (if any). [Note: There is no restriction on the linkage (or absence thereof) of entities that the function body of an exported inline function can reference. A constexpr function (9.1.5) is implicitly inline. — end note]

9.1.7 Type specifiers

The type-specifiers are

1 The type-specifiers are

\[
\begin{align*}
type\text{-}specifier: & \quad \text{simple-type-specifier} \\
& \quad \text{elaborated-type-specifier} \\
& \quad \text{typename\text{-}specifier} \\
& \quad \text{cv\text{-}qualifier} \\
type\text{-}specifier\text{-}seq: & \quad \text{type\text{-}specifier attribute\text{-}specifier\text{-}seq}_{\text{opt}} \\
& \quad \text{type\text{-}specifier type\text{-}specifier\text{-}seq} \\
defining\text{-}type\text{-}specifier: & \quad \text{type\text{-}specifier} \\
& \quad \text{class\text{-}specifier} \\
& \quad \text{enum\text{-}specifier} \\
defining\text{-}type\text{-}specifier\text{-}seq: & \quad \text{defining\text{-}type\text{-}specifier attribute\text{-}specifier\text{-}seq}_{\text{opt}} \\
& \quad \text{defining\text{-}type\text{-}specifier defining\text{-}type\text{-}specifier\text{-}seq}
\end{align*}
\]

The optional attribute\text{-}specifier\text{-}seq in a type\text{-}specifier\text{-}seq or a defining\text{-}type\text{-}specifier\text{-}seq appertains to the type denoted by the preceding type\text{-}specifiers or defining\text{-}type\text{-}specifiers (9.2.3). The attribute\text{-}specifier\text{-}seq affects the type only for the declaration it appears in, not other declarations involving the same type.

2 As a general rule, at most one defining\text{-}type\text{-}specifier is allowed in the complete decl\text{-}specifier\text{-}seq of a declaration or in a defining\text{-}type\text{-}specifier\text{-}seq, and at most one type\text{-}specifier is allowed in a type\text{-}specifier\text{-}seq. The only exceptions to this rule are the following:

\begin{enumerate}
\item const can be combined with any type specifier except itself.
\item volatile can be combined with any type specifier except itself.
\item signed or unsigned can be combined with char, long, short, or int.
\item short or long can be combined with int.
\item long can be combined with double.
\item long can be combined with long.
\end{enumerate}

3 Except in a declaration of a constructor, destructor, or conversion function, at least one defining\text{-}type\text{-}specifier that is not a cv\text{-}qualifier shall appear in a complete type\text{-}specifier\text{-}seq or a complete decl\text{-}specifier\text{-}seq.\footnote{There is no special provision for a decl\text{-}specifier\text{-}seq that lacks a type\text{-}specifier or that has a type\text{-}specifier that only specifies cv\text{-}qualifiers. The “implicit int” rule of C is no longer supported.}

4 [Note: enum\text{-}specifiers, class\text{-}specifiers, and typename\text{-}specifiers are discussed in 9.6, Clause 11, and 13.7, respectively. The remaining type\text{-}specifiers are discussed in the rest of this subclause. — end note]
9.1.7.1 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: 6.7.3 and 9.2.3.5 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 (9.1.1) and its usability in constant expressions (7.7). As described in 9.3, 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 Except that any class member declared mutable (9.1.1) can be modified, any attempt to modify (7.6.19, 7.6.1.5, 7.6.2.2) a const object (6.7.3) during its lifetime (6.6.3) results in undefined behavior. [Example:

```c
const int ci = 3; // cv-qualified (initialized as required)
ci = 4;          // ill-formed: 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;        // ill-formed: 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: modifies a const object
```

For another example,

```c
struct X {
    mutable int i;
    int j;
};
struct Y {
    X x;
    Y();
};

const Y y;
y.x.i++;                     // well-formed: mutable member can be modified
y.x.j++;                     // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y);   // cast away const-ness of y
p->x.i = 99;                 // well-formed: mutable member can be modified
p->x.j = 99;                 // undefined: modifies a const subobject
```

— end example]

5 The semantics of an access through a volatile glvalue are implementation-defined. If an attempt is made to access an object defined with a volatile-qualified type through the use of a non-volatile glvalue, the behavior is undefined.

6 [Note: 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 6.8.1 for detailed semantics. In general, the semantics of volatile are intended to be the same in C++ as they are in C. — end note]

9.1.7.2 Simple type specifiers

The simple type specifiers are

§ 9.1.7.2 161
simple-type-specifier:
  nested-name-specifier_opt type-name
  nested-name-specifier template simple-template-id
decltype-specifier
placeholder-type-specifier
  nested-name-specifier_opt template-name
char
  char8_t
  char16_t
  char32_t
wchar_t
bool
short
int
long
signed
unsigned
float
double
void
type-name:
  class-name
  enum-name
typedef-name

2 A placeholder-type-specifier is a placeholder for a type to be deduced (9.1.7.5). A type-specifier of the form typename_opt nested-name-specifier_opt template-name is a placeholder for a deduced class type (9.1.7.6). The template-name shall name a class template. [Note: An injected-class-name is never interpreted as a template-name in contexts where class template argument deduction would be performed (13.7.1). —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 (6.7.1). Table 12 summarizes the valid combinations of simple-type-specifiers and the types they specify.

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]

9.1.7.3 Elaborated type specifiers [dcl.type.elab]
elaborated-type-specifier:
  class-key attribute-specifier-seq_opt nested-name-specifier_opt identifier
  class-key simple-template-id
  class-key nested-name-specifier template_opt simple-template-id
  enum nested-name-specifier_opt identifier

1 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 (13.8.3), an explicit instantiation (13.8.2) 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: 6.4.4 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 (9.1.7.2). If the identifier or simple-template-id resolves to a typedef-name (9.1.3, 13.2), 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;
Table 12 — *simple-type-specifiers* and the types they specify

<table>
<thead>
<tr>
<th>Specifier(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type-name</code></td>
<td>the type named</td>
</tr>
<tr>
<td><code>simple-template-id</code></td>
<td>the type as defined in 13.2</td>
</tr>
<tr>
<td><code>decltype-specifier</code></td>
<td>the type as defined in 9.1.7.4</td>
</tr>
<tr>
<td><code>placeholder-type-specifier</code></td>
<td>the type as defined in 9.1.7.5</td>
</tr>
<tr>
<td><code>template-name</code></td>
<td>the type as defined in 9.1.7.6</td>
</tr>
<tr>
<td><code>char</code></td>
<td>“char”</td>
</tr>
<tr>
<td><code>unsigned char</code></td>
<td>“unsigned char”</td>
</tr>
<tr>
<td><code>signed char</code></td>
<td>“signed char”</td>
</tr>
<tr>
<td><code>char8_t</code></td>
<td>“char8_t”</td>
</tr>
<tr>
<td><code>char16_t</code></td>
<td>“char16_t”</td>
</tr>
<tr>
<td><code>char32_t</code></td>
<td>“char32_t”</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>“bool”</td>
</tr>
<tr>
<td><code>unsigned</code></td>
<td>“unsigned int”</td>
</tr>
<tr>
<td><code>unsigned int</code></td>
<td>“unsigned int”</td>
</tr>
<tr>
<td><code>signed</code></td>
<td>“int”</td>
</tr>
<tr>
<td><code>signed int</code></td>
<td>“int”</td>
</tr>
<tr>
<td><code>int</code></td>
<td>“int”</td>
</tr>
<tr>
<td><code>unsigned short int</code></td>
<td>“unsigned short int”</td>
</tr>
<tr>
<td><code>unsigned short</code></td>
<td>“unsigned short int”</td>
</tr>
<tr>
<td><code>unsigned long int</code></td>
<td>“unsigned long int”</td>
</tr>
<tr>
<td><code>unsigned long</code></td>
<td>“unsigned long int”</td>
</tr>
<tr>
<td><code>unsigned long long int</code></td>
<td>“unsigned long long int”</td>
</tr>
<tr>
<td><code>unsigned long long</code></td>
<td>“unsigned long long int”</td>
</tr>
<tr>
<td><code>signed long int</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>signed long</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>signed long long</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>long long int</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>long long</code></td>
<td>“long long int”</td>
</tr>
<tr>
<td><code>long int</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>long</code></td>
<td>“long int”</td>
</tr>
<tr>
<td><code>signed short int</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>signed short</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>short int</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>short</code></td>
<td>“short int”</td>
</tr>
<tr>
<td><code>wchar_t</code></td>
<td>“wchar_t”</td>
</tr>
<tr>
<td><code>float</code></td>
<td>“float”</td>
</tr>
<tr>
<td><code>double</code></td>
<td>“double”</td>
</tr>
<tr>
<td><code>long double</code></td>
<td>“long double”</td>
</tr>
<tr>
<td><code>void</code></td>
<td>“void”</td>
</tr>
</tbody>
</table>

is ill-formed. However, the similar declaration `friend T;` is allowed (11.8.3). — end note]

The **class-key** or `enum` keyword present in the *elaborated-type-specifier* shall agree in kind with the declaration to which the name in the *elaborated-type-specifier* refers. This rule also applies to the form of *elaborated-type-specifier* that declares a `class-name` or friend class since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the `enum` keyword shall be used to refer to an enumeration (9.6), the `union class-key` shall be used to refer to a union (Clause 11), and either the `class` or `struct class-key` shall be used to refer to a class (Clause 11) declared using the `class` or `struct class-key`. [Example:

```c
enum class E { a, b };  // OK
enum E x = E::a;
```

— end example]
9.1.7.4 Decltype specifiers

\[ \text{decltype-specifier:} \quad \text{decltype ( expression )} \]

1 For an expression \( e \), the type denoted by `decltype(e)` is defined as follows:

(1.1) if \( e \) is an unparenthesized \( \text{id-expression} \) naming a structured binding (9.5), \( \text{decltype(e)} \) is the referenced type as given in the specification of the structured binding declaration;

(1.2) otherwise, if \( e \) is an unparenthesized \( \text{id-expression} \) naming a non-type \( \text{template-parameter} \) (13.1), \( \text{decltype(e)} \) is the type of the \( \text{template-parameter} \) after performing any necessary type deduction (9.1.7.5, 9.1.7.6);

(1.3) otherwise, if \( e \) is an unparenthesized \( \text{id-expression} \) or an unparenthesized class member access (7.6.1.4), \( \text{decltype(e)} \) 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;

(1.4) otherwise, if \( e \) is a \( \text{xvalue} \), \( \text{decltype(e)} \) is \( T&& \), where \( T \) is the type of \( e \);

(1.5) otherwise, if \( e \) is a \( \text{lvalue} \), \( \text{decltype(e)} \) is \( T& \), where \( T \) is the type of \( e \);

(1.6) otherwise, \( \text{decltype(e)} \) is the type of \( e \).

The operand of the \text{decltype} specifier is an unevaluated operand (7.2).

\[ \text{Example:} \]
\begin{verbatim}
const int&& foo();
int i;
struct A { double x; };  // type is const int&
const A* a = new A();
dectype(foo()) x1 = 17;     // type is const int&
dectype(i) x2;              // type is int
dectype(a->x) x3;           // type is double
dectype((a->x)) x4 = x3;    // type is const double&
\end{verbatim}

\[ \text{— end example} \]

[Note: The rules for determining types involving \text{decltype(auto)} are specified in 9.1.7.5.]

—end note

2 If the operand of a \text{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 \text{decltype-specifier} is analogous to a use of a \text{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]

\[ \text{Example:} \]
\begin{verbatim}
template<class T> struct A { ~A() = delete; };
template<class T> auto h()         // identity
    \rightarrow A<T>;
template<class T> auto i(T)        // forces completion of A<T> and implicitly uses A<T>::~A()
    \rightarrow decltype(h<T>());
    // (A temporary is not introduced as a result of the use of i().)
    \rightarrow void;
template<class T> auto f(T)        // #2
    \rightarrow decltype((h<T>()));
    // OK: calls #2. (#1 is not a viable candidate: type deduction
    // fails (13.9.2) because A<int>::~A() is implicitly used in its
    // decltype-specifier)
auto g() \rightarrow void {
    f(42);
}
template<class T> auto q(T)        // does not force completion of A<T>; A<T>::~A() is not implicitly
    \rightarrow decltype(h<T>());
\end{verbatim}
void r() {
    q(42); // error: deduction against q succeeds, so overload resolution selects
    // the specialization "q(T) -> decltype((h<T>())" with T=int;
    // the return type is &int*, so a temporary is introduced and its
destructor is used, so the program is ill-formed
}

— end example]

9.1.7.5 Placeholder type specifiers

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 in the decl-specifier-seq of a
parameter-declaration of a function declaration or lambda-expression and signifies that the function is an
abbreviated function template (9.2.3.5) or the lambda is a generic lambda (7.5.5).

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 (9.2.3.5), 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 (8.4.1).

4 The type of a variable declared using a placeholder type is deduced from its initializer. This use is allowed in
an initializing declaration (9.3) 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:

    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 z; // error: auto is not a storage-class-specifier
    auto f() -> int; // OK: f returns int
    auto g() { return 0.0; } // OK: g returns double
    auto h(); // OK: h’s return type will be deduced when it is defined

— end example] The auto type-specifier can also be used to introduce a structured binding declaration (9.5).

5 A placeholder type can also be used in the type-specifier-seq in the new-type-id or type-id of a new-expression
(7.6.2.5) and as a decl-specifier of the parameter-declaration’s decl-specifier-seq in a template-parameter
(13.1).

6 A program that uses a placeholder type in a context not explicitly allowed in this subclause is ill-formed.

7 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 (9.1.7.5.1), and if the type
that replaces the placeholder type is not the same in each deduction, the program is ill-formed.

[Example:

    auto x = 5, *y = &x; // OK: auto is int
    auto a = 5, b = { 1, 2 }; // error: different types for auto

— end example]

8 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.

§ 9.1.7.5
If a function with a declared return type that uses a placeholder type has no non-discarded \texttt{return} statements, the return type is deduced as though from a \texttt{return} statement with no operand at the closing brace of the function body. [\textit{Example}:
\begin{verbatim}
    auto f() { } // OK, return type is \texttt{void}
    auto* g() { } // error, cannot deduce \texttt{auto*} from \texttt{void()}
\end{verbatim}
—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 \texttt{private-module-fragment} (if any). [\textit{Note: There is no restriction on the linkage of the deduced return type.} —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 \texttt{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 \texttt{return} statements. [\textit{Example}:
\begin{verbatim}
    auto n = n; // error, \texttt{n}'s initializer refers to \texttt{n}
    auto f();
    void g() { &f; } // error, \texttt{f}'s return type is unknown
    auto sum(int i) {
        if (i == 1)
            return i;
        else
            return sum(i-1)+i; // OK, \texttt{sum}'s return type has been deduced
    }
\end{verbatim}
—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 \texttt{return} statement with a non-type-dependent operand. [\textit{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 (13.9.2).} —end note] [\textit{Example}:
\begin{verbatim}
    template <class T> auto f(T t) { return t; } // return type deduced at instantiation time
    typedef decltype(f(1)) fint_t; // instantiates \texttt{f<int>} to deduce return type
    template<class T> auto f(T* t) { return *t; }
    void g() { int (*p)(int*) = &f; } // instantiates both \texttt{f}s to determine return types, // chooses second
\end{verbatim}
—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. [\textit{Example}:
\begin{verbatim}
    auto f();
    auto f() { return 42; } // return type is \texttt{int}
    auto f(); // OK
    int f(); // error, cannot be overloaded with \texttt{auto f()}
decltype(auto) f(); // error, \texttt{auto} and \texttt{decltype(auto)} don't match
\end{verbatim}
\begin{verbatim}
    template <typename T> auto g(T t) { return t; } // #1
    template auto g(int); // OK, return type is \texttt{int}
    template char g(char); // error, no matching template
    template> auto g(double); // OK, forward declaration with unknown return type
\end{verbatim}
\begin{verbatim}
    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(char); // still matches #1
\end{verbatim}
\begin{verbatim}
    void h() { return g(42); } // error, ambiguous
\end{verbatim}
A function declared with a return type that uses a placeholder type shall not be \texttt{virtual} (11.6.2).

A function declared with a return type that uses a placeholder type shall not be a coroutine (9.4.4).

An explicit instantiation declaration (13.8.2) 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:
\begin{verbatim}
template<typename T> struct A {
  friend T frf(T);
};
auto frf(int i) { return i; }  // not a friend of A<int>

--- end example
\end{verbatim}]

\section*{9.1.7.5.1 Placeholder type deduction}  

\textit{Placeholder type deduction} is the process by which a type containing a placeholder type is replaced by a deduced type.

A type \( T \) containing a placeholder type, and a corresponding initializer \( e \), are determined as follows:

\begin{enumerate}
\item 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}();
\item 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 \textit{braced-init-list} containing only a single \textit{assignment-expression} and \( e \) is the \textit{assignment-expression};
\item for a non-type template parameter declared with a type that contains a placeholder type, \( T \) is the declared type of the non-type template parameter and \( e \) is the corresponding template argument.
\end{enumerate}

In the case of a \texttt{return} statement with no operand or with an operand of type \texttt{void}, \( T \) shall be either \texttt{type-constraint_opt decltype(auto)} or \texttt{cv type-constraint_opt auto}.

If the deduction is for a \texttt{return} statement and \( e \) is a \textit{braced-init-list} (9.3.4), the program is ill-formed.

If the \textit{placeholder-type-specifier} is of the form \texttt{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{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 (13.9.2.1), 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 \( P \). [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 example
\end{verbatim}]

[Example:
\begin{verbatim}
const auto &i = expr;
\end{verbatim}]

The type of \( i \) is the deduced type of the parameter \( u \) in the call \texttt{f(expr)} of the following invented function template:
\begin{verbatim}
template<class U> void f(const U& u);

--- end example
\end{verbatim}]

\§ 9.1.7.5.1
If the placeholder-type-specifier is of the form type-constraint\opt decltype(auto), \( T \) shall be the placeholder alone. The type deduced for \( T \) is determined as described in 9.1.7.2, as though \( e \) had been the operand of the \texttt{decltype}. [Example:

```cpp
int i;
int&& f();
auto x2a(i);
// decltype(x2a) is int
decltype(auto) x2d(i);
// decltype(x2d) is int
auto x3a = 1;
// decltype(x3a) is int
decltype(auto) x3d = 1;
// decltype(x3d) is int
auto x4a = (i);
// decltype(x4a) is int
decltype(auto) x4d = (i);
// decltype(x4d) is int&
auto x5a = f();
// decltype(x5a) is int
decltype(auto) x5d = f();
// decltype(x5d) is int&&
auto x6a = { 1, 2 };  // decltype(x6a) is std::initializer_list<int>
decltype(auto) x6d = { 1, 2 };  // error, { 1, 2 } is not an expression
auto *x7a = &i;
// decltype(x7a) is int*
decltype(auto)*x7d = &i;  // error, declared type is not plain decltype(auto)
```

—end example]

For a placeholder-type-specifier with a type-constraint, if the type deduced for the placeholder does not satisfy its immediately-declared constraint (Clause 13), the program is ill-formed.

9.1.7.6 Deduced class template specialization types

1 If a placeholder for a deduced class type appears as a decl-specifier in the decl-specifier-seq of an initializing declaration (9.3) of a variable, the placeholder is replaced by the return type of the function selected by overload resolution for class template deduction (12.3.1.8). 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 (7.6.2.5), 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 (13.1). A placeholder for a deduced class type shall not appear in any other context.

[Example:

```cpp
template<class T> struct container {
    container(T t) {}
    template<class Iter> container(Iter beg, Iter end);
};
template<class Iter>
container(Iter b, Iter e) -> container<typename std::iterator_traits<Iter>::value_type>;
std::vector<double> v = { /* ... */ };
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]

9.2 Declarators

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.

```cpp
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 (9.11), the specifiers (decl-specifier-seq; 9.1) 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 9.3 and 11.9.

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

```c
T D1, D2, ... Dn;
```

is usually equivalent to

```c
T D1; T D2; ... T Dn;
```

where T is a *decl-specifier-seq* and each D1 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

```c
struct S { /* ... */ };
S S, T; // declare two instances of struct S
```

which is not equivalent to

```c
struct S { /* ... */ };
S S;
S T; // error
```

Another exception is when T is *auto* (9.1.7.5), for example:

```c
auto i = 1, j = 2.0; // error: deduced types for i and j do not match
```

as opposed to

```c
auto i = 1; // OK: i deduced to have type int
auto j = 2.0; // OK: j deduced to have type double
```

—end note]

4 The optional *requires-clause* (Clause 13) in an *init-declarator* or *member-declarator* shall not be present when the declarator does not declare a function (9.2.3.5). When present after a declarator, the *requires-clause* is called the *trailing requires-clause*. The trailing *requires-clause* introduces the *constraint-expression* that results from interpreting its *constraint-logical-or-expression* as a *constraint-expression*. [Example:

```c
void f1(int a) requires true; // OK
auto f2(int a) -> bool requires true; // OK
auto f3(int a) requires true -> bool; // error: requires-clause precedes trailing-return-type
void (*pf)() requires true; // error: constraint on a variable
void g(int (*)(*) requires true); // error: constraint on a parameter-declaration
```

```c
auto* p = new void(*)(char) requires true; // error: not a function declaration
```

—end example]

5 Declarators have the syntax

```c
declarator:
  ptr-declarator
  noptr-declarator parameters-and-qualifiers trailing-return-type
```

```c
ptr-declarator:
  noptr-declarator
  ptr-operator ptr-declarator
```

```c
noptr-declarator:
  declarator-id attribute-specifier-seq attach
  noptr-declarator parameters-and-qualifiers
  noptr-declarator [ constant-expression ] attribute-specifier-seq
  ( ptr-declarator )
```

```c
parameters-and-qualifiers:
  ( parameter-declaration-clause ) cv-qualifier-seq
  ref-qualifier attach
```

```c
trailing-return-type:
  -> type-id
```
ptr-operator:
  • attribute-specifier-seq\opt cv-qualifier-seq\opt
    & attribute-specifier-seq\opt
    && attribute-specifier-seq\opt
  nested-name-specifier • attribute-specifier-seq\opt cv-qualifier-seq\opt

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq\opt

cv-qualifier:
  const
  volatile

ref-qualifier:
  &
  &&

declarator-id:
  ...
  opt id-expression

9.2.1 Type names

To specify type conversions explicitly, and as an argument of \texttt{sizeof}, \texttt{alignof}, \texttt{new}, or \texttt{typeid}, the name of a type shall be specified. This can be done with a \texttt{type-id}, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

\begin{verbatim}
  type-id:
    type-specifier-seq abstract-declarator\opt
  defining-type-id:
    defining-type-specifier-seq abstract-declarator\opt
  abstract-declarator:
    ptr-abstract-declarator
    noptr-abstract-declarator\opt parameters-and-qualifiers trailing-return-type
    abstract-pack-declarator
  ptr-abstract-declarator:
    noptr-abstract-declarator
    ptr-operator ptr-abstract-declarator\opt
  noptr-abstract-declarator:
    noptr-abstract-declarator\opt parameters-and-qualifiers
    noptr-abstract-declarator\opt [ constant-expression\opt ] attribute-specifier-seq\opt
    ( ptr-abstract-declarator )
  abstract-pack-declarator:
    noptr-abstract-pack-declarator
    ptr-operator abstract-pack-declarator
  noptr-abstract-pack-declarator:
    noptr-abstract-pack-declarator parameters-and-qualifiers
    noptr-abstract-pack-declarator [ constant-expression\opt ] attribute-specifier-seq\opt
  ...
\end{verbatim}

It is possible to identify uniquely the location in the \texttt{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. \textit{Example}:

\begin{verbatim}
  int // int i
  int * // int *p
  int *[3] // int *p[3]
  int (*)[3] // int (*)(p3i)[3]
  int *() // int *(f)
  int (*)(double) // int (*)(pf)(double)
\end{verbatim}

name respectively the types “\texttt{int}”, “pointer to \texttt{int}”, “array of 3 pointers to \texttt{int}”, “pointer to array of 3 \texttt{int}”, “function of (no parameters) returning pointer to \texttt{int}”, and “pointer to a function of (\texttt{double}) returning \texttt{int}”. \textit{— end example}]

\footnote{A type can also be named (often more easily) by using a \texttt{typedef} (9.1.3).}
9.2.2 Ambiguity resolution  [dcl.ambig.res]

1 The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 8.8 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 8.8, 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:

```c
struct S {
    S(int);
};

void foo(double a) {
    S w(int(a)); // function declaration
    S x(int()); // function declaration
    S y((int(a))); // object declaration
    S y((int)a); // object declaration
    S z = int(a); // object declaration
}
```
—end example]

2 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)
}
```
—end example]

3 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(*_fp(C_parm[10])); // not: void h(C[10]);
```
—end example]
9.2.3 Meaning of declarators

1 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 (11.3.4, 11.3.7, 11.3.6, 12.5) and for the declaration of template specializations or partial specializations (13.8). 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 (9.7.1)) 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.

Thus, a declaration of a particular identifier has the form

T D

where T is of the form attribute-specifier-seq_opt decl-specifier-seq and D is a declarator. Following is a recursive procedure for determining the type specified for the contained declarator-id by such a declaration.

4 First, the decl-specifier-seq determines a type. In a declaration

T 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” (9.1.7.2). — end example]

In a declaration attribute-specifier-seq_opt T D where D is an unadorned identifier the type of this identifier is “T”.

6 In a declaration T D where D has the form

(D1)

the type of the contained declarator-id is the same as that of the contained declarator-id in the declaration

T D1

Parentheses do not alter the type of the embedded declarator-id, but they can alter the binding of complex declarators.

9.2.3.1 Pointers

1 In a declaration T D where D has the form

* attribute-specifier-seq_opt cv-qualifier-seq_opt D1

and the type of the identifier in the declaration T D1 is “derived-declarator-type-list T”, then the type of the identifier of D is “derived-declarator-type-list cv-qualifier-seq pointer to T”. The cv-qualifiers apply to the pointer and not to the object pointed to. Similarly, the optional attribute-specifier-seq (9.11.1) appertains to the pointer and not to the object pointed to.

[Example: The declarations

const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;

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;]
Examples of ill-formed operations are

```c
ci = 1; // error
ci++; // error
*pc = 2; // error
cp = &ci; // error
cpc++; // error
p = pc; // error
ppc = &p; // error
```

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:

```c
*ppc = &ci; // OK, but would make p point to ci because of previous error
*p = 5; // clobber ci
```

—end example

3 See also 7.6.19 and 9.3.

4 [Note: Forming a pointer to reference type is ill-formed; see 9.2.3.2. Forming a function pointer type is ill-formed if the function type has cv-qualifiers or a ref-qualifier; see 9.2.3.5. Since the address of a bit-field (11.3.9) cannot be taken, a pointer can never point to a bit-field. —end note]

### 9.2.3.2 References

In a declaration `T D` where `D` has either of the forms

```c
& attribute-specifier-seqopt D1
&& attribute-specifier-seqopt 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` (9.1.3, 13.1) or `decltype-specifier` (9.1.7.2), in which case the cv-qualifiers are ignored. [Example:

```c
typedef int& A;
const A aref = 3; // ill-formed; 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.

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;
```

deaclares 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 9.3.3.
—end example

4 It is unspecified whether or not a reference requires storage (6.6.5).

5 There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an initializer (9.3.3) except when the declaration contains an explicit extern specifier (9.1.1), is a class member (11.3) declaration within a class definition, or is the declaration of a parameter or a return type (9.2.3.5); see 6.1. 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 11.3.9, a reference cannot be bound directly to a bit-field. —end note]

6 If a typedef-name (9.1.3, 13.1) or a decltype-specifier (9.1.7.2) 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&&
dectype(r2)& r6 = i; // r6 has the type int&
dectype(r2)&& r7 = i; // r7 has the type int&&
```
—end example]

7 [Note: Forming a reference to function type is ill-formed if the function type has cv-qualifiers or a ref-qualifier; see 9.2.3.5. —end note]

9.2.3.3 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 (9.11.1) 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:

X obj;
// ...
obj.*pmi = 7;   // assign 7 to an integer member of obj
(obj.*pmf)(7);  // call a function member of obj with the argument 7
— end example

A pointer to member shall not point to a static member of a class (11.3.8), 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]

9.2.3.4 Arrays [dcl.array]

1 In a declaration $T D$ where $D$ has the form

\[
T \ [ \ \text{constant-expression}_{\text{opt}} \ ] \ \text{attribute-specifier-seq}_{\text{opt}}
\]

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 an array type; if the type of the identifier of $D$ contains the auto type-specifier, the program is ill-formed. $T$ is called the array element type; this type shall not be a reference type, cv void, or a function type. If the constant-expression (7.7) is present, it shall be a converted constant expression of type std::size_t and its value shall be greater than zero. The constant expression specifies the bound of (number of elements in) the array. If the value of the constant expression is $N$, the array has $N$ elements numbered 0 to $N-1$, and the type of the identifier of $D$ is “derived-declarator-type-list array of $N$ $T$”. An object of array type contains a contiguously allocated non-empty set of $N$ subobjects of type $T$. Except as noted below, if the constant expression is omitted, the type of the identifier of $D$ is “derived-declarator-type-list array of unknown bound of $T$”, an incomplete object type. The type “derived-declarator-type-list array of $N$ $T$” is a different type from the type “derived-declarator-type-list array of unknown bound of $T$”, see 6.7. Any type of the form “cv-qualifier-seq array of $N$ $T$” is adjusted to “array of $N$ cv-qualifier-seq $T$”, and similarly for “array of unknown bound of $T$”. The optional attribute-specifier-seq appertains to the array. [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 $T$” has cv-qualified type; see 6.7.3. — end note]

2 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 another array.

3 When several “array of” specifications are adjacent, a multidimensional array type is created; only the first of the constant expressions that specify the bounds of the arrays may be omitted. 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 (9.2.3.5). An array bound may also be omitted when the declarator is followed by an initializer (9.3), when a declarator for a static data member is followed by a brace-or-equal-initializer (11.3), or in an explicit type conversion (7.6.1.3). In these cases, the bound is calculated from the number of initial elements (say, $N$) supplied (9.3.1), and the type of the identifier of $D$ is “array of $N$ $T$”. 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.

4 [Example:

\[
\text{float fa[17], *afp[17];}
\]

declares an array of float numbers and an array of pointers to float numbers. — end example]
Example:

```c
int x3d[3][5][7];
```

declares an array of three elements, each of which is an array of five elements, each of which is an array of seven integers. The overall array can be viewed as a three-dimensional array of integers, with rank $3 \times 5 \times 7$. Any of the expressions $x3d$, $x3d[i]$, $x3d[i][j]$, $x3d[i][j][k]$ can reasonably appear in an expression. The expression $x3d[i]$ is equivalent to $*(x3d + i)$; in that expression, $x3d$ is subject to the array-to-pointer conversion (7.3.2) and is first converted to a pointer to a 2-dimensional array with rank $5 \times 7$ that points to the first element of $x3d$. Then $i$ is added, which on typical implementations involves multiplying $i$ by the length of the object to which the pointer points, which is $\text{sizeof(int)} \times 5 \times 7$. The result of the addition and indirection is an lvalue denoting the $i$th array element of $x3d$ (an array of five arrays of seven integers). If there is another subscript, the same argument applies again, so $x3d[i][j]$ is an lvalue denoting the $j$th array element of the $i$th array element of $x3d$ (an array of seven integers), and $x3d[i][j][k]$ is an lvalue denoting the $k$th array element of the $j$th array element of the $i$th array element of $x3d$ (an integer). —end example

[Note: The first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations. —end note]

Example:

```c
extern int x[10];
struct S {
    static int y[10];
};
int x[]; // OK: bound is 10
int S::y[]; // OK: bound is 10
void f() {
    extern int x[];
    int i = sizeof(x); // error: incomplete object type
}
```

—end example

[Note: Conversions affecting expressions of array type are described in 7.3.2. —end note]

[Note: The subscript operator can be overloaded for a class (12.5.5). For the operator's built-in meaning, see 7.6.1.1. —end note]

9.2.3.5 Functions [dcl.fct]

In a declaration $T \ D$ where $D$ has the form

```c
D1 ( parameter-declaration-clause ) cv-qualifier-seq
    ref-qualifier opt
    noexcept-specifier 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 noexceptopt function of (parameter-declaration-clause) cv-qualifier-seq opt ref-qualifier_opt returning $T$”, where the optional noexcept is present if and only if the exception specification (14.4) is non-throwing. The optional attribute-specifier-seq appertains to the function type.

2 In a declaration $T \ D$ where $D$ has the form

```c
D1 ( parameter-declaration-clause ) cv-qualifier-seq
    ref-qualifier_opt
    noexcept-specifier opt
    attribute-specifier-seq 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 noexceptopt function of (parameter-declaration-clause) cv-qualifier-seq_opt ref-qualifier_opt returning $U$”, where $U$ is the type specified by the trailing-return-type, and where 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.

3 A type of either form is a function type.\[92\]

\[92\] As indicated by syntax, cv-qualifiers are a significant component in function return types.
parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , parameter-declaration

parameter-declaration:
  attribute-specifier-seq_opt decl-specifier-seq declarator
  attribute-specifier-seq_opt decl-specifier-seq declarator = initializer-clause
  attribute-specifier-seq_opt decl-specifier-seq abstract-declarator_opt
  attribute-specifier-seq_opt decl-specifier-seq abstract-declarator_opt = initializer-clause

The optional attribute-specifier-seq in a parameter-declaration appertains to the parameter.

The parameter-declaration-clause determines the arguments that can be specified, and their processing, when the function is called. [Note: 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. If the parameter-declaration-clause terminates with an ellipsis or a function parameter pack (13.6.3), 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

int printf(const char*, ...);

declares a function that can be called with varying numbers and types of arguments.

  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 <stdarg> 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 (9.1.3, 13.1)) 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 (13.1), or
(6.5) — the type-id of a template-argument for a type-parameter (13.3.1).

[Example:

typedef int FIC(int) const;
FIC f;        // ill-formed: 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:

typedef void F();

§ 9.2.3.5

177
struct S {
    const F f; // OK: equivalent to: void f();
};

—end example]

8 The return type, the parameter-type-list, the ref-qualifier, the cv-qualifier-seq, and the exception specification, but not the default arguments (9.2.3.6) or the trailing requires-clause (9.2), 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]

9 [Example: The declaration
   
   int fseek(FILE*, long, int);

declares a function taking three arguments of the specified types, and returning int (9.1.7). — end example]

10 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 (13.6.6.1).

11 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.

12 Types shall not be defined in return or parameter types.

13 A typedef of function type may be used to declare a function but shall not be used to define a function (9.4). [Example:

    typedef void F();
    F fv; // OK: equivalent to void fv();
    F fv {} // ill-formed
    void fv() {} // OK: definition of fv

    —end example]

14 An identifier can optionally be provided as a parameter name; if present in a function definition (9.4), 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 (6.3.4). — end note]

15 [Example: The declaration

   int i,
   *pi,
   f(),
   *fpi(int),
   (*pif)(const char*, const char*),
   (*fpif(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

    typedef int IFUNC(int);
    IFUNC* fpif(int);

or

    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:

```cpp
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```cpp
template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

—end note

16 A non-template function is a function that is not a function template specialization. [Note: A function template is not a function. — end note]

17 An abbreviated function template is a function declaration whose parameter-type-list includes one or more placeholders (9.1.7.5). An abbreviated function template is equivalent to a function template (13.6.6.1) whose template-parameter-list includes one invented type template-parameter for each occurrence of a placeholder type in the decl-specifier-seq of a parameter-declaration in the function’s parameter-type-list, 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 (9.2.3.5). 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:

```cpp
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.

```cpp
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.

```cpp
template<> void g1<int>(const int*, C2 auto&);
// OK, specialization of g1<int, const double>
```

—end example

18 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:

```cpp
template< typename C = /* ... */>
void g(T x, U y, C auto z);
```

This is functionally equivalent to each of the following two declarations.

```cpp
template< typename T, C U
void g(T x, U y, C auto z);
```

```cpp
template< typename T, C U, C W>
void g(T x, U y, W z);
```

`requires C<U> && C<W>`

```cpp
void g(T x, U y, W z);
```

—end example

19 A function declaration at block scope shall not declare an abbreviated function template.

20 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 (13.6.3). Otherwise, the parameter-declaration is part of a template-parameter-list and declares a template parameter pack; see 13.1. A function parameter pack is a pack expansion (13.6.3). [Example:
template<
    typename... T>
    void f(T (* ...t)(int, int));

int add(int, int);
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.\(^{93}\)

9.2.3.6 Default arguments

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]\(^1\)

[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]\(^2\)

A default argument shall be specified only in the parameter-declaration-clause of a function declaration or lambda-declarator or in a template-parameter (13.1); 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.\(^{94}\)

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. A default argument shall not be redefined by a later declaration (not even to the same value). [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);
    // error: wrong number of arguments
    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
}

93) 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).

94) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or typedef declarations.
void n() {
  f(6); // OK, calls f(6, 7)
}
template<class ... T> struct C {
  void f(int n = 0, T...);
};
C<int> c; // OK, instantiates declaration void C::f(int n = 0, int)

—end example

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; see 6.2. 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 (9.3). 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 13.8.1. [Example: In the following code, g will be called with the value f(2):

```cpp
int a = 1;
int f(int);
int g(int x = f(a)); // default argument: f(::a)

void h() {
  a = 2;
  {
    int a = 3;
    g(); // g(f(::a))
  }
}
—end example]

[Note: In member function declarations, names in default arguments are looked up as described in 6.4.1. Access checking applies to names in default arguments as described in 11.8. —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 (11.3.4.1), copy or move constructor (11.3.4.2), or copy or move assignment operator (11.3.5) 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:

```cpp
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 (6.2) in a default argument. —end note] [Example:

```cpp
void f() {
  int i;
  extern void g(int x = i); // error
  extern void h(int x = sizeof(i)); // OK
  //...
}
—end example]

[Note: The keyword this may not appear in a default argument of a member function; see 7.5.2. [Example:

```cpp
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:

```c
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.

```c
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:

```c
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 (9.8), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also known at any point following the redeclaration where the using-declaration is in scope.

A virtual function call (11.6.2) 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:

```c
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]

9.3 Initializers [dcl.init]

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 (8.6.3), or when an initializer follows a declarator.
initializer:
  brace-or-equal-initializer
  ( expression-list )

brace-or-equal-initializer:
  = initializer-clause
  braced-init-list

initializer-clause:
  assignment-expression
  braced-init-list

braced-init-list:
  { initializer-list , opt }
  { designated-initializer-list , opt }
  { }

initializer-list:
  initializer-clause ... opt
  initializer-list , initializer-clause ... opt

designated-initializer-list:
  designated-initializer-clause
  designated-initializer-list , designated-initializer-clause

designated-initializer-clause:
  designator brace-or-equal-initializer

designator:
  . identifier

eexpr-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 9.1.5, 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:

```c
int f(int);
int a = 2;
int b = f(a);
int c(b);
```
— end example]

3 [Note: Default arguments are more restricted; see 9.2.3.6. — end note]

4 [Note: The order of initialization of variables with static storage duration is described in 6.8.3 and 8.7. — end note]

5 A declaration of a block-scope variable with external or internal linkage that has an initializer is ill-formed.

To zero-initialize an object or reference of type T means:

(6.1) — if T is a scalar type (6.7), the object is initialized to the value obtained by converting the integer literal 0 (zero) to T;\(^{95}\)

(6.2) — if T is a (possibly cv-qualified) non-union class type, its padding bits (6.7) 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 (6.7) 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:

---

\(^{95}\) As specified in 7.3.11, converting an integer literal whose value is 0 to a pointer type results in a null pointer value.
If \( T \) is a (possibly cv-qualified) class type (Clause 11), constructors are considered. The applicable constructors are enumerated (12.3.1.3), and the best one for the initializer (\()\) is chosen through overload resolution (12.3). The constructor thus selected is called, with an empty argument list, to initialize the object.

If \( T \) is an array type, each element is default-initialized.

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

- 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,
- \( T \) is a union with at least one non-static data member, exactly one variant member has a default member initializer,
- \( T \) 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
- each potentially constructed base class of \( T \) is const-default-constructible.

If a program calls for the default-initialization of an object of a const-qualified type \( T \), \( T \) shall be a const-default-constructible class type or array thereof.

To \textit{value-initialize} an object of type \( T \) means:

- if \( T \) is a (possibly cv-qualified) class type (Clause 11) with either no default constructor (11.3.4.1) or a default constructor that is user-provided or deleted, then the object is default-initialized;
- if \( T \) is a (possibly cv-qualified) class type without a user-provided or deleted default constructor, then 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;
- if \( T \) is an array type, then each element is value-initialized;
- otherwise, the object is zero-initialized.

A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

A program whose initializer is an empty set of parentheses, i.e., (\()\), shall be value-initialized.

An object whose initializer is an empty set of parentheses, i.e., (\()\), shall be value-initialized.

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. [\textit{Example}:

```c
int a;

struct X {
    static int a;
    static int b;
};

int X::a = 1;
int X::b = a;  // X::b = X::a
```

If the entity being initialized does not have class type, the \textit{expression-list} in a parenthesized initializer shall be a single expression.
The initialization that occurs in the = form of a brace-or-equal-initializer or condition (8.4), as well as in argument passing, function return, throwing an exception (14.1), handling an exception (14.3), and aggregate member initialization (9.3.1), is called copy-initialization. [Note: Copy-initialization may invoke a move (11.3.4.2). — end note]

The initialization that occurs

(16.1) — for an initializer that is a parenthesized expression-list or a braced-init-list,

(16.2) — for a new-initializer (7.6.2.5),

(16.3) — in a static_cast expression (7.6.1.8),

(16.4) — in a functional notation type conversion (7.6.1.3), and

(16.5) — 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.

(17.1) — If the initializer is a (non-parenthesized) braced-init-list or is = braced-init-list, the object or reference is list-initialized (9.3.4).

(17.2) — If the destination type is a reference type, see 9.3.3.

(17.3) — 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 9.3.2.

(17.4) — If the initializer is O, the object is value-initialized.

(17.5) — Otherwise, if the destination type is an array, the object is initialized as follows. Let x[1], ..., 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 xi for each 1 ≤ i ≤ k, and value-initialized for each k < i ≤ n. For each 1 ≤ i < j ≤ 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.

(17.6) — Otherwise, if the destination type is a (possibly cv-qualified) class type:

(17.6.1) — 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]

(17.6.2) — 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 (12.3.1.3), and the best one is chosen through overload resolution (12.3). Then:

(17.6.2.1) — 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).

(17.6.2.2) — Otherwise, if no constructor is viable, the destination type is a (possibly cv-qualified) aggregate class A, and the initializer is a parenthesized expression-list, the object is initialized as follows. Let e[1], ..., e[n] be the elements of the aggregate (9.3.1). Let x[1], ..., x[k] be the elements of the expression-list. If k is greater than n, the program is ill-formed. The element ei is copy-initialized with xi for 1 ≤ i ≤ k. The remaining elements are initialized with their default member initializers, if any, and otherwise are value-initialized. For each 1 ≤ i < j ≤ n, every value computation and side effect associated with the initialization of ei is sequenced before those associated with the initialization of ej. [Note: By contrast with direct-list-initialization, narrowing conversions (9.3.4) are permitted, designators are not permitted, a temporary object bound to a reference does not have its lifetime extended (6.6.7), and there is no brace elision. [Example:

struct A {
    int a;

```cpp
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 12.3.1.4, and the best one is chosen through overload resolution (12.3). 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 (12.3.1.5), and the best one is chosen through overload resolution (12.3). 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 (13.6.3).
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 (9.3.1) or by an inherited constructor (11.9.3). — 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 (6.6.3). 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 (6.1) of the variable, but the initializing declaration of a non-inline static data member (11.3.8.2) might be the declaration within the class definition and not the definition at namespace scope. — end note]
9.3.1 Aggregates

An aggregate is an array or a class (Clause 11) with

1. no user-declared or inherited constructors (11.3.4),
2. no private or protected non-static data members (11.8),
3. no virtual functions (11.6.2), and
4. no virtual, private, or protected base classes (11.6.1).

[Note: Aggregate initialization does not allow accessing protected and private base class members or constructors. — end note]

The elements of an aggregate are:

1. for an array, the array elements in increasing subscript order, or
2. for a class, the direct base classes in declaration order, followed by the direct non-static data members (11.3) that are not members of an anonymous union, in declaration order.

When an aggregate is initialized by an initializer list as specified in 9.3.4, 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:

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.
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. Otherwise, the initializer list must be \( \{ \} \), and there are no explicitly initialized elements.

For each explicitly initialized element:

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.
```

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 (9.3.4) 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;
  }
}
```
```c
int b3;
struct derived : base1, base2 {
    int d;
};
derived d1{{1, 2}, {}, 4};
derived d2({}, {}, 4);
initializes d1.b1 with 1, d1.b2 with 2, d1.b3 with 42, d1.d with 4, and d2.b1 with 0, d2.b2 with 42, d2.b3 with 42, d2.d with 4. — end example
```

For a non-union aggregate, each element that is not an explicitly initialized element is initialized as follows:

(5.1) If the element has a default member initializer (11.3), the element is initialized from that initializer.

(5.2) Otherwise, if the element is not a reference, the element is copy-initialized from an empty initializer list (9.3.4).

(5.3) Otherwise, the program is ill-formed.

If the aggregate is a union and the initializer list is empty, then

(5.4) if any variant member has a default member initializer, that member is initialized from its default member initializer;

(5.5) otherwise, the first member of the union (if any) is copy-initialized from an empty initializer list.

[Example:

```c
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

```c
struct X { int i, j, k = 42; };
X a[] = { 1, 2, 3, 4, 5, 6 };
X b[2] = { { 1, 2, 3 }, { 4, 5, 6 } };
```

a and b have the same value

```c
struct A {
    string a;
    int b = 42;
    int c = -1;
};
A{.c=21} has the following steps:
```

(5.6) Initialize a with {}
(5.7) Initialize b with = 42
(5.8) Initialize c with = 21

— end example]
```c
```

The initializations of the elements of the aggregate are evaluated in the element order. That is, all value computations and side effects associated with a given element are sequenced before those of any element that follows it in order.

An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 9.3.

The destructor for each element of class type is potentially invoked (11.3.6) 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 (14.2). — end note]

An array of unknown bound initialized with a brace-enclosed initializer-list containing n initializer-clauses is defined as having n elements (9.2.3.4). [Example:

```c
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 \textit{braced-init-list} \{\}.\footnote{The syntax provides for empty \textit{braced-init-list}s, but nonetheless C++ does not have zero length arrays.}  \[\text{Note:} \text{ A default member initializer does not determine the bound for a member array of unknown bound. Since the default member initializer is ignored if a suitable \textit{mem-initializer} is present (11.9.2), the default member initializer is not considered to initialize the array of unknown bound.} \]

\begin{itemize}
  \item \begin{verbatim}
struct S {
  int y[] = { 0 };  // error: non-static data member of incomplete type
};
\end{verbatim}
  \end{itemize}

\begin{itemize}
  \item \begin{verbatim}
} a = { 1, 2, 3 };
\end{verbatim}
  \end{itemize}

Here, the second initializer 2 initializes \texttt{a.j} and not the static data member \texttt{A::s}, and the third initializer 3 initializes \texttt{a.k} and not the unnamed bit-field before it. \[\text{— end example} \] \[\text{— end note} \]

\begin{itemize}
  \item An \textit{initializer-list} is ill-formed if the number of \textit{initializer-clauses} exceeds the number of elements of the aggregate. \[\text{Example:} \]
    \begin{verbatim}
    char cv[4] = { 'a', 's', 'd', 'f', 0 };  // error
    \end{verbatim}
    is ill-formed. \[\text{— end example} \]
  \end{itemize}

\begin{itemize}
  \item 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. \[\text{Example:} \]
    \begin{verbatim}
    struct B {
      int n = B{}.n;
    };
    \end{verbatim}
    \begin{verbatim}
} a1 { A(a,a) };  // OK
    \end{verbatim}
    \begin{verbatim}
const A& a1 { A(a,a) };  // OK
    \end{verbatim}
    \begin{verbatim}
A a(a,a);  // OK
    \end{verbatim}
    \begin{verbatim}
const A& a2 { A{} };  // error
    \end{verbatim}
    \begin{verbatim}
};
    \end{verbatim}
    \begin{verbatim}
struct A {  
  int i;
  static int s;
  int j;
  int :17;
  int k;
};
    \end{verbatim}
    \begin{verbatim}
    struct B {
      int n = B{}.n;
    };
    \end{verbatim}
\begin{verbatim}
};
\end{verbatim}
\[\text{— end example} \]
\end{itemize}

\begin{itemize}
  \item If an aggregate class \texttt{C} contains a subaggregate element \texttt{e} with no elements, the \textit{initializer-clause} for \texttt{e} shall not be omitted from an \textit{initializer-list} for an object of type \texttt{C} unless the \textit{initializer-clauses} for all elements of \texttt{C} following \texttt{e} are also omitted. \[\text{Example:} \]
    \begin{verbatim}
struct S { } s;
struct A {
  S s1;
  int i1;
  S s2;
  int i2;
  S s3;
  int i3;
};
\end{verbatim}
    \begin{verbatim}
} a = {
  { },
  0,
  s,
  0
};
\end{verbatim}
\[\text{— Required initialization} \]
\[\text{— Required initialization} \]
\begin{verbatim}
};
\end{verbatim}
\[\text{— Initialization not required for A::s3 because A::i3 is also not initialized} \]
When initializing a multi-dimensional array, the initializer-clauses initialize the elements with the last (rightmost) index of the array varying the fastest (9.2.3.4). [Example:

```c
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,

```c
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 (11.3.4). Initialization of these aggregate objects is described in 11.9.1. — end note]

[Note: Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 6.8.3.2, 6.8.3.3, and 8.7. — 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;  
    // error
u c = 1;  
u d = { 0, "asdf" };  
    // error
u e = { "asdf" };  
    // error
u f = { .b = "asdf" };  
u g = { .a = 1, .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]

9.3.2 Character arrays

An array of ordinary character type (6.7.1), 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 (9.3).

9.3.3 References

A variable whose declared type is “reference to type T” (9.2.3.2) shall be initialized. [Example:

```c
int g(int) noexcept;  
void f() {
    int i;  
    int& r = i;  
        // r refers to i  
    r = 1;  
        // the value of i becomes 1  
    int* p = &r;  
        // p points to i  
    int& rr = r;  
        // rr refers to what r refers to, that is, to i  
    int (@rg)(int) = g;  
        // rg refers to the function g  
    rg(i);  
        // calls function g  
    int a[3];  
    int (@ra)[3] = a;  
        // ra refers to the array a  
    ra[1] = i;  
        // modifies a[1]
}
```
—end example]

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 (8.6.3) are initializations.

The initializer can be omitted for a reference only in a parameter declaration (9.2.3.5), in the declaration of a function return type, in the declaration of a class member within its class definition (11.3), 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

(5.1.1) — is an lvalue (but not a bit-field), and "cv1 T1" is reference-compatible with "cv2 T2", or

(5.1.2) — has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be converted to an lvalue of type "cv3 T3", where "cv1 T1" is reference-compatible with "cv3 T3" (this conversion is selected by enumerating the applicable conversion functions (12.3.1.6) and choosing the best one through overload resolution (12.3)),

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:

```c
double d = 2.0;
double& rd = d; // rd refers to d
const double& rcd = d; // rcd refers to d

struct A {
};
struct B : A { operator int&(); } b;
A& ra = b; // ra refers to a subobject in b
const & rca = b; // rca refers to a subobject in b
int& ir = B(); // ir refers to the result of B::operator int&
```

— end example]

(5.2) — Otherwise, if the reference is an lvalue reference to a type that is not const-qualified or is volatile-qualified, the program is ill-formed. [Example:

```c
double& rd2 = 2.0; // error: not an lvalue and reference not const
int i = 2;
double& rd3 = i; // error: type mismatch and reference not const
```

— end example]

(5.3) — Otherwise, if the initializer expression

(5.3.1) — is an rvalue (but not a bit-field) or function lvalue and "cv1 T1" is reference-compatible with "cv2 T2", or

(5.3.2) — has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be converted to an rvalue or function lvalue of type "cv3 T3", where "cv1 T1" is reference-compatible with "cv3 T3" (see 12.3.1.6),

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:

```c
struct A {
};
struct B : A { } b;
extern B f();
const & rca2 = f(); // bound to the A subobject of the B rvalue.
A&& rra = f(); // same as above
```

97) This requires a conversion function (11.3.7.2) returning a reference type.
struct X {
   operator B();
   operator int&();
} x;
const A& 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
B& rrb = x;                                      // bound directly to the result of operator B

— end example

(5.4) — Otherwise:

(5.4.1) — If T1 or T2 is a class type and T1 is not reference-related to T2, user-defined conversions are considered using the rules for copy-initialization of an object of type “cv1 T1” by user-defined conversion (9.3, 12.3.1.4, 12.3.1.5); 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.

(5.4.2) — Otherwise, the initializer expression is implicitly converted to a prvalue of type “cv1 T1”. The temporary materialization conversion is applied and the reference is bound to the result.

If T1 is reference-related to T2:

(5.4.3) — cv1 shall be the same cv-qualification as, or greater cv-qualification than, cv2; and

(5.4.4) — if the reference is an rvalue reference, the initializer expression shall not be an lvalue.

[Example:
struct Banana { };  
struct Enigma { operator const Banana(); };  
struct Alaska { operator Banana&(); };  
void enigmatic() {
   typedef const Banana ConstBanana;  
   Banana &&banana1 = ConstBanana();    // ill-formed  
   Banana &&banana2 = Enigma();         // ill-formed  
   Banana &&banana3 = Alaska();         // ill-formed
}

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;                   // error: cv-qualifier dropped
struct A { operator volatile int&(); } a;
const int& r3 = a;                            // error: cv-qualifier dropped
   // from result of conversion function

double d2 = 1.0;                             // error: initializer is lvalue of related type
double& rrd2 = d2;                           // error: initializer is lvalue of related type
struct X { operator int&(); };              
int& rri2 = X();                             // error: result of conversion function is lvalue of related type
int i3 = 2;
double& rrd3 = i3;                           // rrd3 refers to temporary with value 2.0

— end example]

In all cases except the last (i.e., implicitly converting the initializer expression to the referenced type), the reference is said to bind directly to the initializer expression.

6 [Note: 6.6.7 describes the lifetime of temporaries bound to references. — end note]

9.3.4 List-initialization

List-initialization is initialization of an object or reference from a braced-init-list. Such an initializer is called an initializer list, and the comma-separated initializer-clauses of the initializer-list or designated-initializer-clauses of the designated-initializer-list are called the elements of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called direct-list-initialization and list-initialization in a copy-initialization context is called copy-list-initialization. [Note: List-initialization can be used...
N4810

© ISO/IEC

(1.1)

— as the initializer in a variable definition (9.3)

(1.2)

— as the initializer in a new-expression (7.6.2.5)

(1.3)

— in a return statement (8.6.3)

(1.4)

— as a for-range-initializer (8.5)

(1.5)

— as a function argument (7.6.1.2)

(1.6)

— as a subscript (7.6.1.1)

(1.7)

— as an argument to a constructor invocation (9.3, 7.6.1.3)

(1.8)

— as an initializer for a non-static data member (11.3)

(1.9)

— in a mem-initializer (11.9.2)

(1.10)

— 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

3
(3.1)

A constructor is an initializer-list constructor if its first parameter is of type std::initializer_list<E> or
reference to possibly cv-qualified std::initializer_list<E> for some type E, and either there are no other
parameters or else all other parameters have default arguments (9.2.3.6). [Note: Initializer-list constructors
are favored over other constructors in list-initialization (12.3.1.7). 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 (13.9.2.1).
— end note] The template std::initializer_list is not predefined; if the header <initializer_list> is
not included prior to a use of std::initializer_list — even an implicit use in which the type is not
named (9.1.7.5) — the program is ill-formed.
List-initialization of an object or reference of type T is defined as follows:
— 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 (9.3.1).
[Example:
struct A { int x; int y; int z; };
A a{.y = 2, .x = 1};
A b{.x = 1, .z = 2};

// error: designator order does not match declaration order
// 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-listinitialization, 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 appropriatelytyped string literal (9.3.2), initialization is performed as described in that subclause.

(3.4)

— Otherwise, if T is an aggregate, aggregate initialization is performed (9.3.1).
[Example:
double ad[] = { 1, 2.0 };
int ai[] = { 1, 2.0 };

// OK
// error: narrowing

struct S2 {
int m1;
double m2, m3;
};

§ 9.3.4

194


N4810

© ISO/IEC

S2 s21 = { 1, 2, 3.0 };
S2 s22 { 1.0, 2, 3 };
S2 s23 { };

// OK
// error: narrowing
// 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.

(3.7)

— Otherwise, if T is a class type, constructors are considered. The applicable constructors are enumerated
and the best one is chosen through overload resolution (12.3, 12.3.1.7). If a narrowing conversion (see
below) is required to convert any of the arguments, the program is ill-formed.
[Example:
struct S {
S(std::initializer_list<double>);
S(std::initializer_list<int>);
S();
// ...
};
S s1 = { 1.0, 2.0, 3.0 };
S s2 = { 1, 2, 3 };
S s3 = { };

// #1
// #2
// #3
// invoke #1
// invoke #2
// invoke #3

— end example]
[Example:
struct Map {
Map(std::initializer_list<std::pair<std::string,int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};

— end example]
[Example:
struct S {
// no initializer-list constructors
S(int, double, double);
S();
// ...
};
S s1 = { 1, 2, 3.0 };
S s2 { 1.0, 2, 3 };
S s3 { };

// #1
// #2
// OK: invoke #1
// error: narrowing
// OK: invoke #2

— end example]
(3.8)

— Otherwise, if T is an enumeration with a fixed underlying type (9.6), the initializer-list has a single
element v, 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 the underlying type of T, the
program is ill-formed. [Example:
enum
byte
byte
byte
byte

§ 9.3.4

byte : unsigned char { };
b { 42 };
c = { 42 };
d = byte{ 42 };
e { -1 };

//
//
//
//

OK
error
OK; same value as b
error

struct A { byte b; };
A a1 = { { 42 } };
A a2 = { byte{ 42 } };

// error
// OK

void f(byte);
f({ 42 });

// error

195


enum class Handle : uint32_t { Invalid = 0 };  
Handle h { 42 };  // OK

— end example

(3.9) — Otherwise, if the initializer list has a single element of type E and either T is not a reference type or its referenced type is reference-related to E, the object or reference is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization); if a narrowing conversion (see below) is required to convert the element to T, the program is ill-formed.

[Example:
int x1 {2};  // OK
int x2 {2.0};  // error: narrowing
— end example]

(3.10) — Otherwise, if T is a reference type, a prvalue of the type referenced by T is generated. The prvalue initializes its result object by copy-list-initialization. The prvalue is then used to direct-initialize the reference. [Note: As usual, the binding will fail and the program is ill-formed if the reference type is a lvalue reference to a non-const type. — end note]

[Example:
struct S {
    S(std::initializer_list<double>);  // #1
    S(const std::string&);  // #2
    // ...
};
const Sk r1 = { 1, 2, 3.0 };  // OK: invoke #1
const Sk r2 { "Spinach" };  // OK: invoke #2
Sk r3 = { 1, 2, 3 };  // error: initializer is not an lvalue
const int& i1 = { 1 };  // OK
const int& i2 = { 1.1 };  // error: narrowing
const int (&iar)[2] = { 1, 2 };  // OK: iar is bound to temporary array

struct A { }; a;
struct B { explicit B(const A&); };  
const B &b2(a);  // error: cannot copy-list-initialize B temporary from A
— end example]

(3.11) — Otherwise, if the initializer list has no elements, the object is value-initialized.

[Example:
int** pp {};  // initialized to null pointer
— end example]

(3.12) — Otherwise, the program is ill-formed.

[Example:
struct A { int i; int j; };  
A a1 { 1, 2 };  // aggregate initialization
A a2 { 1.2 };  // error: narrowing
struct B {
    B(std::initializer_list<int>);  
};
B b1 { 1, 2 };  // creates initializer_list<int> and calls constructor
B b2 { 1, 2.0 };  // error: narrowing
struct C {
    C(int i, double j);  
};
C c1 = { 1, 2.2 };  // calls constructor with arguments (1, 2.2)
C c2 = { 1.1, 2 };  // error: narrowing
int j { 1 };  // initialize to 1
int k { };  // initialize to 0
— end example]
4 Within the *initializer-list* of a *braced-init-list*, the *initializer-clauses*, including any that result from pack expansions (13.6.3), 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]

5 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 shall be accessible (11.8) 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:

```cpp
struct X {
    X(std::initializer_list<double> v);
};
X x{ 1,2,3 };`n
```

The initialization will be implemented in a way roughly equivalent to this:

```cpp
const double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

assuming that the implementation can construct an *initializer_list* object with a pair of pointers. — end example]

6 The array has the same lifetime as any other temporary object (6.6.7), 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:

```cpp
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f() {
    std::vector<cmplx> v2{ 1, 2, 3 };
    std::initializer_list<int> i3 = { 1, 2, 3 };
}

struct A {
    std::initializer_list<int> i4;
    A() : i4{ 1, 2, 3 } {} // ill-formed, would create a dangling reference
};
```

For *v1* and *v2*, the *initializer_list* object is a parameter in a function call, so the array created for { 1, 2, 3 } has full-expression lifetime. For *i3*, the *initializer_list* object is a variable, so the array persists for the lifetime of the variable. For *i4*, the *initializer_list* object is initialized in the constructor’s ctor-initializer as if by binding a temporary array to a reference member, so the program is ill-formed (11.9.2). — 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]

7 A narrowing conversion is an implicit conversion

(7.1) — from a floating-point type to an integer type, or

(7.2) — from *long* *double* to *double* or *float*, or from *double* to *float*, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

(7.3) — from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

(7.4) — from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type.

§ 9.3.4
[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 u11 = {-1}; // error: narrows
signed int a11 =
{ (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
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]

9.4 Function definitions

9.4.1 In general

Function definitions have the form

```
function-definition:
  attribute-specifier-seqopt decl-specifier-seqopt declarator virt-specifier-seqopt function-body
```

```
function-body:
  ctor-initializeropt 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` (11.3).

In a `function-definition`, either `void declarator` or `declarator` shall be a well-formed function declaration as described in 9.2.3.5. 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 an incomplete or abstract (possibly cv-qualified) class type in the context of the function definition unless the function is deleted (9.4.3).

[Example: A simple example of a complete function definition is

```c
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 11.3.4 and 11.9.

[Note: A `cv-qualifier-seq` affects the type of `this` in the body of a member function; see 9.2.3.2. — end note]

[Note: Unused parameters need not be named. For example,

```c
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 6.3.3).
The function-local predefined variable \texttt{__func__} is defined as if a definition of the form

\begin{verbatim}
static const char __func__[] = "function-name";
\end{verbatim}

had been provided, where \texttt{function-name} is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.\footnote{Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (5.10). If a predefined variable is not odr-used (6.2), its string value need not be present in the program image.}

\begin{verbatim}
Example:
\end{verbatim}

\begin{verbatim}
struct S {
    S() : s(__func__) { } // OK
    const char* s;
};
void f(const char* s = __func__); // error: __func__ is undeclared
\end{verbatim}

\section*{9.4.2 Explicitly-defaulted functions} \[dcl.fct.def.default\]

A function definition whose \texttt{function-body} is of the form \texttt{= default ;} is called an \textit{explicitly-defaulted} definition. A function that is explicitly defaulted shall

\begin{enumerate}
    \item be a special member function or a comparison operator (7.6.8, 7.6.9, 7.6.10), and
    \item not have default arguments.
\end{enumerate}

The type \(T_1\) of an explicitly defaulted 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 \texttt{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 would have been implicitly declared as \texttt{constexpr}. If a function is explicitly defaulted on its first declaration, it is implicitly considered to be \texttt{constexpr} if the implicit declaration would be.

\begin{verbatim}
Example:
\end{verbatim}

\begin{verbatim}
struct S {
    constexpr S() = default; // ill-formed: implicit S() is not constexpr
    S(int a = 0) = default; // ill-formed: default argument
    void operator=(const S&) = default; // ill-formed: 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
\end{verbatim}

\begin{verbatim}
struct T {
    T();
    T(T&&) noexcept(false);
};
struct U {
    T t;
    U();
    U(U&&) noexcept = default; // OK: defines copy constructor
};
U u1;
U u2 = static_cast<U&&>(u1); // OK, calls std::terminate if T::T(T&&) throws
\end{verbatim}
Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (11.3.4 11.3.6, 11.3.4.2, 11.3.5), which might mean defining them 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:**

```
struct trivial {
    trivial() = default;
    trivial(const trivial&) = default;
    trivial(trivial&&) = default;
    trivial& operator=(const trivial&) = default;
    trivial& operator=(trivial&&) = default;
    ~trivial() = default;
};
struct nontrivial1 {
    nontrivial1();
};
nontrivial1::nontrivial1() = default; // not first declaration
```

**9.4.3 Deleted definitions**

A function definition whose *function-body* is of the form `= delete ;` is called a *deleted definition*. A function with a deleted definition is also called a *deleted function*.

A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [Note: 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 (6.2) of a virtual function does not, by itself, constitute a reference. — end note]

**Example:** One can prevent default initialization and initialization by non-`double`s with

```
struct onlydouble {
    onlydouble() = delete; // OK, but redundant
    template<class T>
    onlydouble(T) = delete;
    onlydouble(double);
};
```

**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.

```
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[]
```

**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.

```
struct moveonly {
    moveonly() = default;
    moveonly(const moveonly&) = default;
```
moveonly(moveonly&&) = default;
moveonly& operator=(const moveonly&) = delete;
moveonly& operator=(moveonly&&) = default;
~moveonly() = default;
};
moveonly* p;
moveonly q(*p); // error, deleted copy constructor
—end example

4 A deleted function is implicitly an inline function (9.1.6). [Note: The one-definition rule (6.2) 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 (6.6.5.4) shall not be defined as deleted. [Example:

struct sometype {
    sometype();
};
sometype::sometype() = delete; // ill-formed; not first declaration
—end example]

9.4.4 Coroutine definitions

dcl.fct.def.coroutine

1 A function is a coroutine if it contains a coroutine-return-statement (8.6.3.1), 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.

2 [Example:

task<int> f();

task<void> g1() {
    int i = co_await f();
    std::cout << "f() => " << i << std::endl;
}

template <typename... Args>
task<void> g2(Args&&...)
    { // OK, ellipsis is a pack expansion
        int i = co_await f();
        std::cout << "f() => " << i << std::endl;
    }

task<void> g3(int a, ...)
    { // error: variable parameter list not allowed
        int i = co_await f();
        std::cout << "f() => " << i << std::endl;
    }
—end example]

3 The promise type of a coroutine is std::coroutine_traits<R, 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 (12.3.1) if the coroutine is a non-static member function. The promise type shall be a class type.

4 In the following, pᵢ is an lvalue of type Pᵢ, where pᵢ denotes *this and pᵢ₊₁ denotes the iᵗʰ function parameter for a non-static member function, and pᵢ denotes the iᵗʰ function parameter otherwise.

5 A coroutine behaves as if its function-body were replaced by:
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) — promise-type denotes the promise type, and
(5.4) — the object denoted by the exposition-only name promise is the promise object of the coroutine, and
(5.5) — the label denoted by the name final-suspend is defined for exposition only (8.6.3.1), and
(5.6) — 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 (12.3.2), 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. \[Note: If the unqualified-id return_void is found, flowing off the end of a coroutine is equivalent to a co_return with no operand. Otherwise, flowing off the end of a coroutine results in undefined behavior (8.6.3.1). \] —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 (6.6.5.4.1). 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 (12.3.2), 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 (6.4.5). 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;
```

§ 9.4.4
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;
}

—end example

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 (6.6.5.4.2). 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, a copy is created for each coroutine parameter. Each such copy is an object with automatic storage duration that is direct-initialized from an lvalue referring to the corresponding parameter if the parameter is an lvalue reference, and from an xvalue referring to it otherwise. A reference to a parameter in the function-body of the coroutine and in the call to the coroutine promise constructor is replaced by a reference to its copy. 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]

14 If the evaluation of the expression promise.unhandled_exception() exits via an exception, the coroutine is considered suspended at the final suspend point.

15 The expression co_await promise.final_suspend() shall not be potentially-throwing (14.4).
9.5 Structured binding declarations

A structured binding declaration introduces the identifiers \( v_0, v_1, v_2, \ldots \) of the identifier-list as names (6.3.1) 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). 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

\[
\text{attribute-specifier-seq}\opt S \ cv \ A \ e;
\]

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

\[
\text{attribute-specifier-seq}\opt \text{decl-specifier-seq}\opt \text{ref-qualifier}\opt e \ \text{initializer};
\]

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:

\[
\text{auto f()} \to \text{int}(&)[2];
\]

\[
\text{auto [ x, y ] = f();}
\]

// \( x \) and \( y \) refer to elements in a copy of the array return value

\[
\text{auto& [ xr, yr ] = f();}
\]

// \( xr \) and \( yr \) refer to elements in the array referred to by \( f \)'s return value

— end example]

Otherwise, if the qualified-id \( \text{std::tuple_size}\opt E\opt::\text{value} \) names a complete class type with a member named value, the expression \( \text{std::tuple_size}\opt E\opt::\text{value} \) 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 \( \text{std::size_t} \) corresponding to \( v_i \). The unqualified-id \( \text{get} \) is looked up in the scope of \( E \) by class member access lookup (6.4.5), 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 \( e.\text{get}\langle i\rangle() \). Otherwise, the initializer is \( \text{get}\langle i\rangle(e) \), where \( \text{get} \) is looked up in the associated namespaces (6.4.2). In either case, \( \text{get}\langle i\rangle \) is interpreted as a template-id. [Note: Ordinary unqualified lookup (6.4.1) 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 \( \text{std::tuple_element}\opt i\opt, E\opt::\text{type} \) and the type \( U_j \) designated by either \( T_j \& \) or \( T_j\&\& \), where \( U_j \) is an lvalue reference if the initializer is an lvalue and an rvalue reference otherwise, variables are introduced with unique names \( r_i \) as follows:

\[
S \ U_j \ r_i = \text{initializer};
\]

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 \( e.\text{name} \) 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 \( m_0, m_1, m_2, \ldots \) (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:

\[
\text{struct S { int x1 : 2; volatile double y1; };}
\]

\[
S f();
\]

\[
\text{const auto [ x, y ] = f();}
\]

The type of the id-expression \( x \) is "\text{const int}" , the type of the id-expression \( y \) is "\text{const volatile double}". — end example]

9.6 Enumeration declarations

An enumeration is a distinct type (6.7.2) with named constants. Its name becomes an enum-name within its scope.

\[
\text{enum-name: identifier}
\]

© ISO/IEC N4810

§ 9.6
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-nameSpecifier, 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:

```c
struct S {
    enum E : int {
        enum E : int {};
        enum E : int {};
        // error: redeclaration of enumeration
    }
};
```]

If the enum-head-name of an opaque-enum-declaration contains a nested-name-specifier, the declaration shall be an explicit specialization (13.8.3).

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:

```c
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```]
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.

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 (9.7.1) 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:

- 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.

- If no initializer is specified for the first enumerator, its type is an unspecified signed integral type.

- 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 (6.3.2)
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 not 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 0.99

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
```

99) This set of values is used to define promotion and conversion semantics for the enumeration type. It does not preclude an
expression of enumeration type from having a value that falls outside this range.
Note that this implicit `enum` to `int` conversion is not provided for a scoped enumeration:

```cpp
enum class Col { red, yellow, green }

int x = Col::red; // error: no Col to int conversion
Col y = Col::red;
if (y) {} // error: no Col to bool conversion
```

—end example

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. These names obey the scope rules defined for all names in 6.3 and 6.4. [Example:

```cpp
enum direction { left='l', right='r' }

void g() {
    direction d; // OK
    d = left; // OK
    d = direction::right; // OK
}

enum class altitude { high='h', low='l' }

void h() {
    altitude a; // OK
    a = high; // error: high not in scope
    a = altitude::low; // OK
}
```

—end example] 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:

```cpp
struct X {
    enum direction { left='l', right='r' }
    int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p) {
    direction d; // error: direction not in scope
    int i;
    i = p->f(left); // error: left not in scope
    i = p->f(X::right); // OK
    i = p->f(p->left); // OK
    // ...
}
```

—end example]

### 9.7 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.

[Note: A namespace name with external linkage is exported if any of its namespace-definitions is exported, or if it contains any export-declarations (10.2). A namespace is never attached to a module, and never has module linkage even if it is not exported. —end note] [Example:

```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]

The outermost declarative region of a translation unit is a namespace; see 6.3.6.
9.7.1 Namespace definition  

`namespace-name:
  identifier
namespace-alias`

`namespace-definition:
  named-name-space-definition
  unnamed-name-space-definition
  nested-name-space-definition`

`named-name-space-definition:
  `inline_opt` namespace attribute-specifier-seq_opt identifier { namespace-body }

unnamed-name-space-definition:
  `inline_opt` namespace attribute-specifier-seq_opt { namespace-body }

`nested-name-space-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 in the global scope or in a namespace scope (6.3.6).

2 In a `named-name-space-definition`, the `identifier` is the name of the namespace. If the `identifier`, when looked up (6.4.1), refers to a `namespace-name` (but not a `namespace-alias`) that was introduced in the namespace in which the `named-name-space-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-name-space-definition` appears.

3 Because a `namespace-definition` contains declarations in its `namespace-body` and a `namespace-definition` is itself a declaration, it follows that `namespace-definition`s can be nested. [Example:

```c
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; }  // Outer::i
    int i;
    void g() { i++; }  // Inner::i
  }
}
```

—end example]

4 The enclosing namespaces of a declaration are those namespaces in which the declaration lexically appears, except for a redeclaration of a namespace member outside its original namespace (e.g., a definition as specified in 9.7.1.2). Such a redeclaration has the same enclosing namespaces as the original declaration. [Example:

```c
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]

5 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 (6.4.2) whenever one of them is, and a using-directive (9.7.3) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (9.7.1.1). Furthermore, each member of the inline namespace can subsequently be partially specialized (13.6.5), explicitly instantiated (13.8.2), or explicitly specialized (13.8.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (6.4.3.2) 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

\[
\text{namespace } E \{ \text{inline opt namespace } I \{ B \} \}
\]

where the optional inline is present if and only if the identifier \( I \) is preceded by inline. [Example:

```c
namespace A {inline B::C {
  int i;
}}
```

The above has the same effect as:

```c
namespace A {
  inline namespace B {
    namespace C {
      int i;
    }
  }
}
```

—end example]

### 9.7.1.1 Unnamed namespaces

An unnamed-namespace-definition behaves as if it were replaced by

\[
\text{inline opt namespace unique \{ /* empty body */ \}}
\]

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:

```c
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
}
```
9.7.1.2 Namespace member definitions

A declaration in a namespace \( N \) (excluding declarations in nested scopes) whose `declarator-id` is an `unqualified-id` (9.2.3), whose `class-head-name` (Clause 11) or `enum-head-name` (9.6) is an identifier, or whose `elaborated-type-specifier` is of the form `class-key attribute-specifier-seq_opt identifier` (9.1.7.3), or that is an `opaque-enum-declaration`, declares (or redeclares) its `unqualified-id` or `identifier` as a member of \( N \). [Note: An explicit instantiation (13.8.2) or explicit specialization (13.8.3) 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()
```

— end example]

Members of a named namespace can also be defined outside that namespace by explicit qualification (6.4.3.2) 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 R {
    void Q::V::g() { /* ... */ } // error: R doesn’t enclose Q
}
```

— end example]

3 If a friend declaration in a non-local class first declares a class, function, class template or function template the friend is a member of the innermost enclosing namespace. The friend declaration does not by itself make the name visible to unqualified lookup (6.4.1) or qualified lookup (6.4.3). [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 (6.4.2). 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 X {
        friend void f(X);  // A::f(X) is a friend
        class Y {
            friend void g();   // A::g is a friend
            friend void h(int);  // A::h is a friend
            friend void f2<>(int);  // ::f2<>(int) is a friend
        };
    };
}

// A::f, A::g and A::h are not visible here
X x;
void g() { f(x); }   // definition of A::g
void f(X) { /* ... */ }  // definition of A::f
void h(int) { /* ... */ }  // definition of A::h

// A::f, A::g and A::h are visible here and known to be friends
using A::x;
void h() {
    A::f(x);
    A::X::f(x);   // error: f is not a member of A::X
    A::X::Y::g();  // error: g is not a member of A::X::Y
}

—end example]

9.7.2 Namespace alias

A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

```
namespace-alias-definition:
    namespace identifier = qualified-namespace-specifier ;

qualified-namespace-specifier:
    nested-name-specifier opt namespace-name
```

1 The identifier in a namespace-alias-definition is a synonym for the name of the namespace denoted by the qualified-namespace-specifier and becomes a namespace-alias. [Note: When looking up a namespace-name in a namespace-alias-definition, only namespace names are considered, see 6.4.6. — end note]

2 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:

```cpp
namespace Company_with_very_long_name { /* ... */ }
namespace CVVLN = Company_with_very_long_name;
namespace CVVLN = Company_with_very_long_name;  // OK: duplicate
namespace CVVLN = CVVLN;

—end example]
```

9.7.3 Using directive

```
using-directive:
    attribute-specifier-seq opt using namespace nested-name-specifier opt namespace-name ;
```

1 A using-directive shall not appear in class scope, but may appear in namespace scope or in block scope. [Note: When looking up a namespace-name in a using-directive, only namespace names are considered, see 6.4.6. — end note] The optional attribute-specifier-seq appertains to the using-directive.

2 A using-directive specifies that the names in the nominated namespace can be used in the scope in which the using-directive appears after the using-directive. During unqualified name lookup (6.4.1), 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]

3 A using-directive does not add any members to the declarative region in which it appears. [Example:

```c
namespace A {
  int i;
  namespace B {
    namespace C {
      int i;
    }
    using namespace A::B::C;
    void f1() {
      i = 5;   // OK, C::i visible in B and hides A::i
    }
  }
  namespace D {
    using namespace B;
    using namespace C;
    void f2() {
      i = 5;   // ambiguous, B::C::i or A::i?
    }
    void f3() {
      i = 5;   // uses A::i
    }
    void f4() {
      i = 5;   // ill-formed; neither i is visible
    }
  }
}
```

—end example]

4 For unqualified lookup (6.4.1), 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 6.4.3.2. — 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
    }
  }
```

§ 9.7.3 212
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]

5 If a namespace is extended (9.7.1) 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.

6 [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 (6.4). 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,

```cpp
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]

7 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.101 [Example:
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(1);  // error: ambiguous: D::f(int) or E::f(int)?
f('a');  // OK: D::f(char)
}

—end example]

### 9.8 The using declaration

[namespace.udecl]

```
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
```

Each using-declarator in a using-declaration\(^1\) 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 (6.4.3, 11.7) 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:

```c
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’s nested-name-specifier shall name a base class of the class being defined. If a using-declarator names a constructor, its nested-name-specifier shall name a direct base class of the class being defined. [Example:

```c
template <typename... bases>
struct X : bases...
  using bases::g...;
};
```

\(1\) A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using-declarator each.
X<B, D> x; // OK: B::g and D::g introduced

— end example] [Example:

    class C {
        int g();
    };

    class D2 : public B {
        using B::f;          // OK: B is a base of D2
        using B::e;          // OK: e is an enumerator of base B
        using B::x;          // OK: x is a union member of base B
        using C::g;          // error: C isn’t a base of D2
    };

— end example]

4 [Note: 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 (13.6.2). — end note] If a constructor or assignment operator brought from a base class into a derived class has the signature of a copy/move constructor or assignment operator for the derived class (11.3.4.2, 11.3.5), 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:

    struct A {
        template <class T> void f(T);
        template <class T> struct X { };
    };

    struct B : A {
        using A::f<double>;     // ill-formed
        using A::X<int>;        // ill-formed
    };

— end example]

6 A using-declaration shall not name a namespace.

7 A using-declaration shall not name a scoped enumerator.

8 A using-declaration that names a class member shall be a member-declaration. [Example:

    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]

9 Members declared by a using-declaration can be referred to by explicit qualification just like other member names (6.4.3.2). [Example:

    void f();

    namespace A {
        void g();
    }

    namespace X {
        using ::f;        // global f
        using A::g;        // A’s g
    }
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 {   // OK: double declaration
    using A::i, A::i;
}

struct B {  
    int i;
};

struct X : B {  // error: double member declaration
    using B::i, B::i;
};
```

—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 (9.2.3.6), default template arguments (13.1), and template specializations (13.6.5, 13.8.3) are considered. —end note] [Example:

```cpp
namespace A {  
    void f(int);
}

using A::f;       // f is a synonym for A::f; that is, for A::f(int).

namespace A {  
    void f(char);
}

void foo() {  // calls f(int), even though f(char) exists.
    f('a');
}

void bar() {   // f is a synonym for A::f; that is, for A::f(int) and A::f(char).
    using A::f;
    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 (13.6.5). —end note]

Since a `using-declaration` is a declaration, the restrictions on declarations of the same name in the same declarative region (6.3) also apply to `using-declarations`. [Example:

```cpp
namespace A {  
    int x;
}
```
namespace B {
    int i;
    struct g {
    }
    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 ] — end note]  

If a function declaration in namespace scope or block scope has the same name and the same parameter-
type-list (9.2.3.5) 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, return type, and template parameter list 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'); // calls C::f(char)
    f(1); // error: ambiguous: B::f(int) or C::f(int)?
    void f(int); // error: f(int) conflicts with C::f(int) and B::f(int)
}

— end example ] — end note]

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 (9.2.3.5), 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);

§ 9.8 217
void h(int);
};

struct D : B {
    using B::f;
    void f(int); // OK: D::f(int) overrides B::f(int);
    using B::g;
    void g(char); // OK
    using B::h;
    void h(int); // OK: D::h(int) hides B::h(int)
};

void k(D* p)
{
    p->f(1); // calls D::f(int)
    p->f('a'); // calls B::f(char)
    p->g(1); // calls B::g(int)
    p->g('a'); // calls D::g(char)
}

struct B1 {
    B1(int);
};

struct B2 {
    B2(int);
};

struct D1 : B1, B2 {
    using B1::B1;
    using B2::B2;
};
D1 d1(0); // ill-formed: ambiguous

struct D2 : B1, B2 {
    using B1::B1;
    using B2::B2;
    D2(int); // OK: D2::D2(int) hides B1::B1(int) and B2::B2(int)
};
D2 d2(0); // calls D2::D2(int)

—end example

[Note: 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 (11.7). In particular, the implicit object parameter is treated as if it were a reference to the derived class rather than to the base class (12.3.1). 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]

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 (6.4.3.1) or forming a set of overload candidates (12.3.1.3, 12.3.1.4, 12.3.1.7). [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 (11.9.3). A constructor of a derived class is sometimes preferred to a constructor of a base class if they would otherwise be ambiguous (12.3.3). — end note]

In a using-declarator that does not name a constructor, all members of the set of introduced declarations shall be accessible. In a using-declarator that names a constructor, no access check is performed. In particular, if a derived class uses a using-declarator to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible. The base class members mentioned by a using-declarator shall be visible in the scope of at least one of the direct base classes of the class where the using-declarator is specified.

§ 9.8
[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 {
    using A::x;
    int x(int);
};

struct C : A {
    using A::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]

20 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:

```
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]

21 If a using-declarator uses the keyword typename and specifies a dependent name (13.7.2), the name introduced by the using-declaration is treated as a typedef-name (9.1.3).

9.9 The asm declaration

1 An asm declaration has the form

```
asm-definition:
    attribute-specifier-seq_optasm ( string-literal )
```

The asm declaration is conditionally-supported; its meaning is implementation-defined. The optional attribute-specifier-seq in an asm-definition appertains to the asm declaration. [Note: Typically it is used to pass information through the implementation to an assembler. —end note]

9.10 Linkage specifications

1 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 (6.5) between C++ and non-C++ code fragments can be achieved using a linkage-specification:
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]

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]

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 (6.3). 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" typedef void FUNC();
FUNC f2; // the name f2 has C++ language linkage and the function's type has C language linkage
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
}
```
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]

5 If two declarations declare functions with the same name and parameter-type-list (9.2.3.5) 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.

6 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 6.2); this implies that such an entity must not be defined in more than one namespace  scope. — end note] [Example:

```c
int x;
namespace A {
    extern "C" int f();
    extern "C" int g() { return 1; }
    extern "C" int h();
    extern "C" int x();  // ill-formed: 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; }  // ill-formed, 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; }  // definition for the function h with C language linkage  // A::h and ::h refer to the same function
—end example]
```

7 A declaration directly contained in a linkage-specification is treated as if it contains the extern specifier (9.1.1)  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:

```c
extern "C" double f();
static double f();  // error
extern "C" int i;  // declaration
```
extern "C" { // definition
  int i;
}
extern "C" static void g(); // error
—end example]

8 [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 value refers is considered a C function. — end note]

9 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.

9.11 Attributes [dcl.attr]

9.11.1 Attribute syntax and semantics [dcl.attr.grammar]

1 Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

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-namespace 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:
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 (13.6.3). 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 (6.4) 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, 9.2). 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 (11.8.3), that declaration shall be a definition. No **attribute-specifier-seq** shall appertain to an explicit instantiation (13.8.2).

6 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. [Note: Each implementation should choose a distinctive name for the **attribute-namespace** in an **attribute-scoped-token**. — end note]

7 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]  

---

### 9.11.2 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** (14.3). An **alignment-specifier** may also be applied to the declaration of a class (in an **elaborated-type-specifier** (9.1.7.3) or **class-head** (Clause 11), respectively). An **alignment-specifier** with an ellipsis is a pack expansion (13.6.3).

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 (6.6.6), or evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed.

An **alignment-specifier** of the form **alignas(type-id)** has the same effect as **alignas(alignof(type-id))** (7.6.2.7).

The alignment requirement of an entity is the strictest nonzero alignment specified by its **alignment-specifiers**, if any; otherwise, the **alignment-specifiers** have no effect.
The combined effect of all alignment-specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all alignment-specifiers appertaining to that entity were omitted. [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]

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; // error: definition of S lacks alignment, no diagnostic required
    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:

```c
alignas(T) alignas(A) T buffer[N];
```
Specifying alignas(T) ensures that the final requested alignment will not be weaker than alignof(T), and therefore the program will not be ill-formed. —end example]

[Example:

```c
alignas(double) void f(); // error: alignment applied to function
alignas(double) unsigned char c[sizeof(double)]; // array of characters, suitably aligned for a double
extern unsigned char c[sizeof(double)]; // no alignas necessary
alignas(float)
    extern unsigned char c[sizeof(double)]; // error: different alignment in declaration
```
—end example]

### 9.11.3 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 (6.8.2) 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:

```c
/** Translation unit A. */

struct foo { int* a; int* b; }
std::atomic<struct foo *> foo_head[10];
```

§ 9.11.3 224
int foo_array[10][10];

[[carries_dependency]] struct foo* f(int i) {
    return foo_head[i].load(memory_order::consume);
}

int g(int* x, int* y [[carries_dependency]]) {
    return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries_dependency]] struct foo* f(int i);
int g(int* x, int* y [[carries_dependency]]);

int c = 3;

void h(int i) {
    struct foo* p;
    p = f(i);
    do_something_with(g(&c, p->a));
    do_something_with(g(p->a, &c));
}

The carries_dependency attribute on function f means that the return value carries a dependency out of f, so that the implementation need not constrain ordering upon return from f. Implementations of f and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences). Function g’s second parameter has a carries_dependency attribute, but its first parameter does not. Therefore, function h’s first call to g carries a dependency into g, but its second call does not. The implementation might need to insert a fence prior to the second call to g. — end example

9.11.4 Contract attributes

9.11.4.1 Syntax

Contract attributes are used to specify preconditions, postconditions, and assertions for functions.

contract-attribute-specifier:
    [[ [ expects contract-levelopt : conditional-expression ] ]
    [[ ensures contract-levelopt identifieropt : conditional-expression ] ]
    [[ assert contract-levelopt : conditional-expression ] ]

contract-level:
    default
    audit
    axiom

An ambiguity between a contract-level and an identifier is resolved in favor of contract-level.

A contract-attribute-specifier using expects is a precondition. It expresses a function’s expectation on its arguments and/or the state of other objects using a predicate that is intended to hold upon entry into the function. The attribute may be applied to the function type of a function declaration.

A contract-attribute-specifier using ensures is a postcondition. It expresses a condition that a function should ensure for the return value and/or the state of objects using a predicate that is intended to hold upon exit from the function. The attribute may be applied to the function type of a function declaration. A postcondition may introduce an identifier to represent the glvalue result or the prvalue result object of the function. When the declared return type of a non-templated function contains a placeholder type, the optional identifier shall only be present in a definition. [Example:

int f(char * c)
    [[ensures res: res > 0 & c != nullptr]]; \n
int g(double * p)
    [[ensures audit res: res != 0 & p != nullptr & p <= 0.0]];
A contract-attribute-specifier using `assert` is an assertion. It expresses a condition that is intended to be satisfied where it appears in a function body. The attribute may be applied to a null statement (8.2). An assertion is checked by evaluating its predicate as part of the evaluation of the null statement it applies to.

Preconditions, postconditions, and assertions are collectively called contracts. The conditional-expression in a contract is contextually converted to `bool` (7.3); the converted expression is called the predicate of the contract. [Note: The predicate of a contract is potentially evaluated (6.2). — end note]

The only side effects of a predicate that are allowed in a contract-attribute-specifier are modifications of non-volatile objects whose lifetime began and ended within the evaluation of the predicate. An evaluation of a predicate that exits via an exception invokes the function `std::terminate` (14.5.1). The behavior of any other side effect is undefined. [Example:

```cpp
void push(int x, queue & q)
    [[expects: !q.full()]]
    [[ensures: !q.empty()]]
{
    /* ... */
    [[assert: q.is_valid()]];
    /* ... */
}
```

—end example]
The predicate of a contract condition has the same semantic restrictions as if it appeared as the first expression-statement in the body of the function it applies to, except that the return type of the function is known in a contract condition appertaining to its definition, even if the return type contains a placeholder type.

A precondition is checked by evaluating its predicate immediately before starting evaluation of the function body. [Note: The function body includes the function-try-block (Clause 14) and the ctor-initializer (11.9.2).—end note] A postcondition is checked by evaluating its predicate immediately before returning control to the caller of the function. [Note: The lifetime of local variables and temporaries has ended. Exiting via an exception or via longjmp (17.13.2) is not considered returning control to the caller of the function. —end note]

If a function has multiple preconditions, their evaluation (if any) will be performed in the order they appear lexically. If a function has multiple postconditions, their evaluation (if any) will be performed in the order they appear lexically. [Example:

```c
void f(int * p)
    [[expects: p != nullptr]]    // #1
    [[ensures: *p == 1]]        // #3
    [[expects: *p == 0]]        // #2
{
    *p = 1;
}

—end example]
```

If a postcondition odr-uses (6.2) a non-reference parameter in its predicate and the function body makes direct or indirect modifications of the value of that parameter, the behavior is undefined. [Example:

```c
int f(int x)
    [[ensures r: r == x]]
{
    return ++x;
    // undefined behavior
}

void g(int * p)
    [[ensures: p != nullptr]]
{
    *p = 42;
    // OK, p is not modified
}

int h(int x)
    [[ensures r: r == x]]
{
    potentially_modify(x);
    // undefined behavior if x is modified
    return x;
}

—end example]
```

9.11.4.3 Checking contracts [dcl.attr.contract.check]

If the contract-level of a contract-attribute-specifier is absent, it is assumed to be default. [Note: A default contract-level is expected to be used for those contracts where the cost of run-time checking is assumed to be small (or at least not expensive) compared to the cost of executing the function. An audit contract-level is expected to be used for those contracts where the cost of run-time checking is assumed to be large (or at least significant) compared to the cost of executing the function. An axiom contract-level is expected to be used for those contracts that are formal comments and are not evaluated at run-time. —end note]

[Note: Multiple contract conditions may be applied to a function type with the same or different contract-levels. [Example:

```c
int z;
```
bool is_prime(int k);

void f(int x)
    [[expects: x > 0]]
    [[expects audit: is_prime(x)]]
    [[ensures: z > 10]]
{
    /* ... */
}

—end example — end note

A translation may be performed with one of the following build levels: off, default, or audit. A translation with build level set to off performs no checking for any contract. A translation with build level set to default performs checking for default contracts. A translation with build level set to audit performs checking for default and audit contracts. If no build level is explicitly selected, the build level is default. The mechanism for selecting the build level is implementation-defined. The translation of a program consisting of translation units where the build level is not the same in all translation units is conditionally-supported. There should be no programmatic way of setting, modifying, or querying the build level of a translation unit.

During constant expression evaluation (7.7), only predicates of checked contracts are evaluated. In other contexts, it is unspecified whether the predicate for a contract that is not checked under the current build level is evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined.

The violation handler of a program is a function of type “noexcept opt function of (lvalue reference to const std::contract_violation) returning void”. The violation handler is invoked when the predicate of a checked contract evaluates to false (called a contract violation). There should be no programmatic way of setting or modifying the violation handler. It is implementation-defined how the violation handler is established for a program and how the std::contract_violation (17.8.2) argument value is set, except as specified below. If a precondition is violated, the source location of the violation is implementation-defined. [Note: Implementations are encouraged but not required to report the caller site. — end note] If a postcondition is violated, the source location of the violation is the source location of the function definition. If an assertion is violated, the source location of the violation is the source location of the statement to which the assertion is applied.

If a violation handler exits by throwing an exception and a contract is violated on a call to a function with a non-throwing exception specification, then the behavior is as if the exception escaped the function body. [Note: The function std::terminate is invoked (14.5.1). — end note] [Example:

    void f(int x) noexcept [[expects: x > 0]];  

    void g() {  
    f(0);  
    }  

    —end example]

A translation may be performed with one of the following violation continuation modes: off or on. A translation with violation continuation mode set to off terminates execution by invoking the function std::terminate (14.5.1) after completing the execution of the violation handler. A translation with a violation continuation mode set to on continues execution after completing the execution of the violation handler. If no continuation mode is explicitly selected, the default continuation mode is off. [Note: A continuation mode set to on provides the opportunity to install a logging handler to instrument a pre-existing code base and fix errors before enforcing checks. — end note] [Example:

    void f(int x) [[expects: x > 0]];  

    void g() {  
    f(0);  
    }  

    —end example]
9.11.5 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:

```
( string-literal )
```

[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]

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.

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.

[Note: Implementations may 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 may include the text provided within the `attribute-argument-clause` of any `deprecated` attribute applied to the name or entity. — end note]

9.11.6 Fallthrough attribute

The attribute-token `fallthrough` may be applied to a null statement (8.2); 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 (8.4.2). 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. The program is ill-formed if there is no such statement.

[Note: The use of a fallthrough statement is intended to 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. — end note]

[Example:

```c
void f(int n) {
  void g(), h(), i();
  switch (n) {
    case 1:
    case 2:
      g();
      [[fallthrough]];  // warning on fallthrough discouraged
    case 3:
      h();
    case 4:
      i();
      [[fallthrough]];  // ill-formed
  }
}
```
—— end example]

9.11.7 Likelihood attributes

The attribute-tokens `likely` and `unlikely` may be applied to labels or statements. The `attribute-tokens` `likely` and `unlikely` shall appear at most once in each `attribute-list` and no `attribute-argument-clause` shall be present. The `attribute-token` `likely` shall not appear in an `attribute-specifier-seq` that contains the `attribute-token` `unlikely`.

[Note: 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...]

§ 9.11.7
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. Excessive usage of either of these attributes is liable to result in performance degradation. — end note

3 [Example:
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]

9.11.8 Maybe unused attribute [dcl.attr.unused]
1 The attribute-token maybe_unused indicates that a name or entity is possibly intentionally unused. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present.
2 The attribute may be applied to the declaration of a class, a typedef-name, a variable (including a structured binding declaration), a non-static data member, a function, an enumeration, or an enumerator.
3 For an entity marked maybe_unused, implementations should not emit a warning that the entity or its structured bindings (if any) are used or unused. For a structured binding declaration not marked maybe_unused, implementations should not emit such a warning unless all of its structured bindings are unused.
4 A name or entity declared without the maybe_unused attribute can later be redeclared with the attribute and vice versa. An entity is considered marked after the first declaration that marks it.
5 [Example:
[[maybe_unused]] void f([[maybe_unused]] bool thing1,
    [[maybe_unused]] bool thing2) {
    [[maybe_unused]] bool b = thing1 && thing2;
    assert(b);
}
Implementations should not warn that b is unused, whether or not NDEBUG is defined. — end example]

9.11.9 Nodiscard attribute [dcl.attr.nodiscard]
1 The attribute-token nodiscard may be applied to the declarator-id in a function declaration or to the declaration of a class or enumeration. It shall appear at most once in each attribute-list and no attribute-argument-clause shall be present.
2 [Note: A nodiscard call is a function call expression that calls a function previously declared nodiscard, or whose return type is a possibly cv-qualified class or enumeration type marked nodiscard. Appearance of a nodiscard call as a potentially-evaluated discarded-value expression (7.2) is discouraged unless explicitly cast to void. Implementations should issue a warning in such cases. This is typically because discarding the return value of a nodiscard call has surprising consequences. — end note]
3 [Example:
struct [[nodiscard]] error_info { /* ... */ }; error_info enable_missile_safety_mode(); void launch_missiles();]
9.11.10 Noreturn attribute

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] [Note: Implementations should issue a warning if a function marked `[[noreturn]]` might return. — end note]

3 [Example:
```c
[[ noreturn ]] void f() {
    throw "error";  // OK
}
```
```c
[[ noreturn ]] void q(int i) {
    // behavior is undefined if called with an argument <= 0
    if (i > 0)
        throw "positive";
}
```
— end example]

9.11.11 No unique address attribute

The attribute-token `no_unique_address` specifies that a non-static data member is a potentially-overlapping subobject (6.6.2). 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:
```c
template<typename Key, typename Value,
        typename Hash, typename Pred, typename Allocator>
class hash_map {
    [[no_unique_address]] Hash hasher;
    [[no_unique_address]] Pred pred;
    [[no_unique_address]] Allocator alloc;
    Bucket *buckets;
    // ...
    public:
    // ...
};
```
Here, `hasher`, `pred`, and `alloc` could have the same address as `buckets` if their respective types are all empty. — end example]
10 Modules

10.1 Module units and purviews

module-declaration:
  export_opt module module-name module-partition_opt attribute-specifier-seq_opt ;

module-name:
  module-name-qualifier_opt identifier

module-partition:
  : module-name-qualifier_opt identifier

module-name-qualifier:
  identifier .

module-name-qualifier identifier .

1 A module unit is a translation unit that contains a module-declaration. A named module is the collection of module units with the same module-name. The identifiers module and import shall not appear as identifiers in a module-name or module-partition. The optional attribute-specifier-seq appertains to the module-declaration.

2 A module interface unit is a module unit whose module-declaration contains the export keyword; any other module unit is a module implementation unit. A named module shall contain exactly one module interface unit with no module-partition, known as the primary module interface unit of the module; no diagnostic is required.

3 A module partition is a module unit whose module-declaration contains a module-partition. A named module shall not contain multiple module partitions with the same module-partition. All module partitions of a module that are module interface units shall be directly or indirectly exported by the primary module interface unit (10.3). No diagnostic is required for a violation of these rules. [Note: Module partitions can be imported only by other module units in the same module. The division of a module into module units is not visible outside the module. — end note]

4 [Example:

Translation unit #1:
  export module A;
  export import :Foo;
  export int baz();

Translation unit #2:
  export module A:Foo;
  import :Internals;
  export int foo() { return 2 * (bar() + 1); }

Translation unit #3:
  module A:Internals;
  int bar();

Translation unit #4:
  module A;
  import :Internals;
  int bar() { return baz() - 10; }
  int baz() { return 30; }

Module A contains four translation units:

(4.1) — a primary module interface unit,
(4.2) — a module partition A:Foo, which is a module interface unit forming part of the interface of module A,
(4.3) — a module partition A:Internals, which does not contribute to the external interface of module A, and
(4.4) — a module implementation unit providing a definition of bar and baz, which cannot be imported because it does not have a partition name.

§ 10.1
A module unit purview is the sequence of tokens starting at the module-declaration and extending to the end of the translation unit. The purview of a named module M is the set of module unit purviews of M’s module units.

The global module is the collection of all global-module-fragments and all translation units that are not module units. Declarations appearing in such a context are said to be in the purview of the global module. [Note: The global module has no name, no module interface unit, and is not introduced by any module-declaration. — end note]

A module is either a named module or the global module. A declaration is attached to a module as follows:

1. If the declaration
   - is a replaceable global allocation or deallocation function (17.6.2.1, 17.6.2.2), or
   - is a namespace-declaration with external linkage, or
   - appears within a linkage-specification,
     it is attached to the global module.

2. Otherwise, the declaration is attached to the module in whose purview it appears.

A module-declaration that contains neither export nor a module-partition implicitly imports the primary module interface unit of the module as if by a module-import-declaration. [Example:

Translation unit #1:
```c
module B:Y;
int y();
```

// does not implicitly import B

Translation unit #2:
```c
export module B;
import :Y;
int n = y();
```

// OK, does not create interface dependency cycle

Translation unit #3:
```c
module B:X1;
int &a = n;
```

// does not implicitly import B

// error: n not visible here

Translation unit #4:
```c
module B:X2;
import B;
int &b = n;
```

// OK

Translation unit #5:
```c
module B;
int &c = n;
```

// implicitly imports B

// OK

— end example]

10.2 Export declaration [module.interface]

```c
export-declaration
export declaration
export { declaration-seq_opt }
```

An export-declaration shall appear only at namespace scope and only in the purview of a module interface unit. An export-declaration shall not appear directly or indirectly within an unnamed namespace or a private-module-fragment. An export-declaration has the declarative effects of its declaration or its declaration-seq (if any). An export-declaration does not establish a scope and its declaration or declaration-seq shall not contain an export-declaration.

A declaration is exported if it is

- a namespace-scope declaration declared within an export-declaration, or
- a module-import-declaration declared with the export keyword (10.3), or
- a namespace-definition that contains an exported declaration, or
- a declaration within a header unit (10.3) that introduces at least one name.
The *interface* of a module `M` is the set of all exported declarations within its purview. [Example:

```c
export module M;
namespace A { // exported
    export int f(); // exported
    int g(); // not exported
}
```

The interface of `M` comprises `A` and `A::f`. — end example]

3 An exported 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 // purview of a module interface unit
export module M;
export namespace {} // error: does not introduce any names
export namespace {
    int a1; // error: export of name with internal linkage
}
namespace {
    export int a2; // error: export of name with internal linkage
}
export static int b; // error: b explicitly declared static
export int f(); // OK
export namespace N {} // OK
export using namespace N; // error: does not declare a name
--end example]
```

5 If the declaration is a *using-declaration* (9.8) 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
```
Note: These constraints do not apply to type names introduced by typedef declarations and alias-declarations. [Example:

```
export module M;
struct S;
export using T = S;  // OK, exports name T denoting type S
```
— end example]  — end note]

6 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:

```
export module M;
struct S { int n; };
typedef S S;
export typedef S S;  // OK, does not redeclare an entity
export struct S;  // error: exported declaration follows non-exported declaration
```
— end example]

7 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 6.5. Namespace-scope names exported by a module are visible to name lookup in any translation unit importing that module; see 6.3.6. 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:

```
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;  // & exported as incomplete
export auto rootFinder(double a) {
    return [=](double x) { return (x + a/x)/2; };
}
export const int n = 5;  // OK, n has external linkage
```

Implementation unit of M:

```
module M;
struct A {
    int value;
};
```

Main program:

```
import M;
int main() {
    X::f();  // OK, X is exported and definition of X is reachable
    X::X y;
    auto f = rootFinder(2);  // OK
    return A(45).value;  // error: A is incomplete
}
```

— end example]

8 [Note: Redeclaring a name in an export-declaration cannot change the linkage of the name (6.5). [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;
namespace {
    namespace N {
        extern int x;  // #3
    }
    export int N::x;  // error: #3 gives internal linkage
    static_assert(1 == 1);  // error: does not declare a name
}
```

—end example] — end note]

9 [Note: Declarations in an exported namespace-definition or in an exported linkage-specification (9.10) are exported and subject to the rules of exported declarations. [Example:

```c
export module M;
export namespace N {
    int x;  // OK
    static_assert(1 == 1);  // error: does not declare a name
}
```

—end example] — end note]

### 10.3 Import declaration

**[module.import]**

```c
module-import-declaration:
    export opt import module-name attribute-specifier-seq_opt ;
    export opt import module-partition attribute-specifier-seq_opt ;
    export opt import header-name attribute-specifier-seq_opt ;
```

1 In a module unit, all module-import-declarations shall precede all other top-level-declarations in the top-level-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 (6.3.6) in the importing translation unit and declarations within the imported translation units become reachable (10.6) 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). — end note] An importable header is a member of an implementation-defined set of headers. H shall identify an importable header. Two module-import-declarations import the same header unit if and only if their header-names identify the same header or source file (15.2). [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 15.4. — end note] A declaration of a name with internal linkage is permitted within a header unit despite all declarations being implicitly exported (10.2). If such a declaration declares an entity that is odr-used outside the header unit, or by a template instantiation whose point of instantiation is outside the header unit, the program is ill-formed.

6 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. When a module-import-declaration in a module unit imports another module unit of the same module, it also imports all translation units imported by all module-import-declarations in that module unit. These rules may in turn lead to the importation of yet more translation units.

7 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]

8 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]

9 A translation unit has an interface dependency on a module unit U if it contains a module-declaration or module-import-declaration that imports U or if it has an interface dependency on a module unit that has an interface dependency on U. A translation unit shall not have an interface dependency on itself. [Example:
Interface unit of M1:
  export module M1;
  import M2;

Interface unit of M2:
  export module M2;
  import M3;

Interface unit of M3:
  export module M3;
  import M1;  // error: cyclic interface dependency M3 → M1 → M2 → M3
— end example]

10.4 Global module fragment

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. [Note: Prior to phase 4 of translation, only preprocessing directives can appear in the global module fragment (15.3). — end note]

2 A declaration D is decl-reachable from a declaration S in the same translation unit if:

(2.1) — D does not declare a function or function template and S contains an id-expression, namespace-name, type-name, template-name, or concept-name naming D, or

(2.2) — D declares a function or function template that is named by an expression (6.2) appearing in S, or

(2.3) — S contains an expression E of the form
        postfix-expression ( expression-listopt )

whose postfix-expression denotes a dependent name, or for an operator expression whose operator denotes a dependent name, and D is found by name lookup for the corresponding name in an expression synthesized from E by replacing each type-dependent argument or operand with a value of a placeholder type with no associated namespaces or entities, or

(2.4) — S contains an expression that takes the address of an overloaded function (12.4) whose set of overloads contains D and for which the target type is dependent, or

(2.5) — there exists a declaration M that is not a namespace-definition for which M is decl-reachable from S and either

(2.5.1) — D is decl-reachable from M, or

(2.5.2) — D redeclares the entity declared by M or M redeclares the entity declared by D, and D is neither a friend declaration nor a block-scope declaration, or

(2.5.3) — D declares a namespace N and M is a member of N, or

§ 10.4
(2.5.4) — one of \(M\) and \(D\) declares a class or class template \(C\) and the other declares a member or friend of \(C\), or

(2.5.5) — one of \(D\) and \(M\) declares an enumeration \(E\) and the other declares an enumerator of \(E\), or

(2.5.6) — \(D\) declares a function or variable and \(M\) is declared in \(D\),\(^{103}\) or

(2.5.7) — one of \(M\) and \(D\) declares a template and the other declares a partial or explicit specialization or an implicit or explicit instantiation of that template, or

(2.5.8) — one of \(M\) and \(D\) declares a class or enumeration type and the other introduces a typedef name for linkage purposes for that type.

In this determination, it is unspecified

(2.6) — 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,

(2.7) — 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,

(2.8) — whether a decltype-specifier that does not denote a dependent type is replaced by its denoted type prior to this determination, and

(2.9) — 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 top-level-declaration in the top-level-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 (13.7.4.1) are outside the module unit, even when the instantiation context (10.5) includes the module unit. — end note]

4 [Example:

```cpp
const int size = 2;
int ary1[size]; // unspecified whether size is decl-reachable from ary1
constexpr int identity(int x) { return x; }
int ary2[identity(2)]; // unspecified whether identity is decl-reachable from ary2

template<typename> struct S;
template<typename, int> struct S2;
constexpr int g(int);
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]

5 [Example:

Source file "foo.h":

```cpp
namespace N {
  struct X {};
  int d();
  int e();
  inline int f(X, int = d()) { return e(); }
  int g(X);
  int h(X);
}
```

Module \(M\) interface:

```cpp
module;
#include "foo.h"
export module M;
```

\(^{103}\) A declaration can appear within a lambda-expression in the initializer of a variable.
template<typename T> int use_f() {
    N::X x;                          // N::X, N, and :: are decl-reachable from use_f
    return f(x, 123);               // N::t is decl-reachable from use_f,
                                      // N::e is indirectly decl-reachable from use_f
                                      // because it is decl-reachable from N::f, and
                                      // N::d is decl-reachable from use_f
                                      // because it is decl-reachable from N::f
                                      // even though it is not used in this call
}

template<typename T> int use_g() {
    N::X x;                          // N::X, N, and :: are decl-reachable from use_f
    return g((T()), x);             // N::g is not decl-reachable from use_g
}

template<typename T> int use_h() {
    N::X x;                          // N::X, N, and :: are decl-reachable from use_h
    return h((T()), x);             // N::h is not decl-reachable from use_h, but
                                      // N::h is decl-reachable from use_h<int>
}

int k = use_h<int>();           // use_h<int> is decl-reachable from k, so
                                // N::h is decl-reachable from k

Module M implementation:

module M;
    int a = use_f<int>();          // OK
    int b = use_g<int>();          // error: no viable function for call to g;
                                    // g is not decl-reachable from purview of
                                    // module M’s interface, so is discarded
    int c = use_h<int>();          // OK
—end example]

10.5 Instantiation context

The instantiation context is a set of points within the program that determines which names are visible to argument-dependent name lookup (6.4.2) and which declarations are reachable (10.6) in the context of a particular declaration or template instantiation.

2 During the implicit definition of a defaulted special member function (11.3.3), 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 special member function.

3 During the implicit instantiation of a template whose point of instantiation is specified as that of an enclosing specialization (13.7.4.1), 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 top-level-declaration-seq of the primary module interface unit of M (prior to the private-module-fragment, if any).

4 During the implicit instantiation of a template that is implicitly instantiated because it is referenced from within the implicit definition of a defaulted special member function, the instantiation context is the instantiation context of the defaulted special member function.

5 During the instantiation of any other template specialization, the instantiation context comprises the point of instantiation of the template.

6 In any other case, the instantiation context at a point within the program comprises that point.

7 [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 {};

§ 10.5 239
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
- (7.1) the point at the end of translation unit #1,
- (7.2) the point at the end of translation unit #2, and
- (7.3) the point of the call to f(0),
so the definition of X is reachable (10.6).

It is unspecified whether the call to g(0) is valid: the instantiation context of bar<int, X> comprises
- (7.4) the point at the end of translation unit #1,
- (7.5) the point at the end of translation unit #3, and
- (7.6) the point of the call to g(0),
so the definition of X is not necessarily reachable, as described in 10.6. — end example]

10.6 Reachability [module.reach]

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 (6.3.6). — 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.\(^{104}\) [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 or necessarily reachable, respectively, if, for any point \( P \) in the instantiation context (10.5),
- (3.1) \( D \) appears prior to \( P \) in the same translation unit, or
- (3.2) \( D \) is not discarded (10.4), appears in a translation unit that is reachable or necessarily reachable from \( P \), respectively, and either does not appear within a private-module-fragment or appears in a private-module-fragment of the module containing \( P \).

[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,

\(^{104}\) Implementations are therefore not required to prevent the semantic effects of additional translation units involved in the compilation from being observed.
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;
   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]

5 [Note: An entity can have reachable declarations even if it is not visible to name lookup. — end note]

[Example:

   export module A;
   struct X {};
   export using Y = X;

   module B;
   import A;
   Y y;                        // OK, definition of X is reachable
   X x;                        // ill-formed: X not visible to unqualified lookup

—end example]
11 Classes

A class is a type. Its name becomes a class-name (11.2) within its scope.

```
class-name:
    identifier
    simple-template-id
```

A class-specifier or an elaborated-type-specifier (9.1.7.3) 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-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
```

A class declaration where the class-name in the class-head-name is a simple-template-id shall be an explicit specialization (13.8.3) or a partial specialization (13.6.5). 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 (9.7.1) 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.

4 If a class is marked with the class-virt-specifier final and it appears as a class-or-decltype in a base-clause (11.6), 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
```
—end example]

5 [Note: Complete objects of class type have nonzero size. Base class subobjects and members declared with the no_unique_address attribute (9.11.11) are not so constrained. — end note]
6 [Note: Class objects can be assigned (7.6.19, 12.5.3, 11.3.5), passed as arguments to functions (9.3, 11.3.4.2), and returned by functions (except objects of classes for which copying or moving has been restricted; see 9.4.3 and 11.8). Other plausible operators, such as equality comparison, can be defined by the user; see 12.5. — end note]

7 A \textit{union} is a class defined with the \textit{class-key union}; it holds at most one data member at a time (11.4). [Note: Aggregates of class type are described in 9.3.1. — end note]

11.1 \textbf{Properties of classes} \hskip 2em [\textit{class.prop}]

1 A \textit{trivially copyable class} is a class:

\begin{itemize}
  \item[(1.1)] where each copy constructor, move constructor, copy assignment operator, and move assignment operator (11.3.4.2, 11.3.5) is either deleted or trivial,
  \item[(1.2)] that has at least one non-deleted copy constructor, move constructor, copy assignment operator, or move assignment operator, and
  \item[(1.3)] that has a trivial, non-deleted destructor (11.3.6).
\end{itemize}

2 A \textit{trivial class} is a class that is trivially copyable and has one or more default constructors (11.3.4.1), all of which are either trivial or deleted and at least one of which is not deleted. [Note: In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. — end note]

3 A class $\mathcal{S}$ is a \textit{standard-layout class} if it:

\begin{itemize}
  \item[(3.1)] has no non-static data members of type non-standard-layout class (or array of such types) or reference,
  \item[(3.2)] has no virtual functions (11.6.2) and no virtual base classes (11.6.1),
  \item[(3.3)] has the same access control (11.8) for all non-static data members,
  \item[(3.4)] has no non-standard-layout base classes,
  \item[(3.5)] has at most one base class subobject of any given type,
  \item[(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
  \item[(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.\footnote{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).} [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]
    \begin{itemize}
      \item[(3.7.1)] If $\mathcal{X}$ is a non-union class type with no (possibly inherited (11.6)) non-static data members, the set $M(\mathcal{X})$ is empty.
      \item[(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 $M(\mathcal{X}_0)$.
      \item[(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 $\mathcal{X}$.
      \item[(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 $M(\mathcal{X}_e)$.
      \item[(3.7.5)] If $\mathcal{X}$ is a non-class, non-array type, the set $M(\mathcal{X})$ is empty.
    \end{itemize}
\end{itemize}

4 [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 { };
struct S : Q { };
struct T : Q { };
struct U : S, T { }; // not a standard-layout class
```

— end example]
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 11.3. — 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]

11.2 Class names

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]

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 (6.3). 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 (6.4.4). [Example:

```cpp
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 11.8.3, operator functions in 12.5. —end example] —end note]

3 [Note: An elaborated-type-specifier (9.1.7.3) 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.

11.3 Class members [class.mem]

member-specification:
member-declaration member-specification_opt
access-specifier : member-specification_opt
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 (11.4.1) and direct members thereof. Members of a class are data members, member functions (11.3.1), nested types, enumerators, and member templates (13.6.2) 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]

A member-declaration does not declare new members of the class if it is

1. a friend declaration (11.8.3),
2. a static_assert-declaration,
3. a using-declaration (9.8), or
4. an empty-declaration.

For any other member-declaration, each declared entity that is not an unnamed bit-field (11.3.9) 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.

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 (11.2, 11.3.10) and enumerations (9.6) declared in the class and arbitrary types declared as members by use of a typedef declaration (9.1.3) or alias-declaration.

The enumerators of an unscoped enumeration (9.6) defined in the class are members of the class.

A data member or member function may be declared static in its member-declaration, in which case it is a static member (see 11.3.8) (a static data member (11.3.8.2) or static member function (11.3.8.1), 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 (11.3.2), respectively). [Note: A non-static data member of non-reference type is a member subobject of a class object (6.6.2). — end note]

A member shall not be declared twice in the member-specification, except that

1. a nested class or member class template can be declared and then later defined, and
2. an enumeration can be introduced with an opaque-enum-declaration and later redeclared with an enum-specifier.

§ 11.3
[Note: A single name can denote several member functions provided their types are sufficiently different (Clause 12). — end note]

6 A complete-class context of a class is a

- function body (9.4.1),
- default argument (9.2.3.6),
- noexcept-specifier (14.4),
- contract condition (9.11.4), or
- 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 (6.7) (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:

```c
struct S {
    using T = void();
    T * p = 0;     // OK: brace-or-equal-initializer
    virtual T f = 0; // OK: pure-specifier
};
```

— end example]

9 In a member-declarator for a bit-field, the constant-expression is parsed as the longest sequence of tokens that could syntactically form a constant-expression. [Example:

```c
int a;
const int b = 0;
struct S {
    int x1 : 8 = 42;       // OK, "*= 42" is brace-or-equal-initializer
    int x2 : 8 { 42 };    // OK, "( 42 )" is brace-or-equal-initializer
    int y1 : true ? 8 : a = 42;  // OK, brace-or-equal-initializer is absent
    int y2 : true ? 8 : b = 42;  // error: cannot assign to const int
    int y3 : (true ? 8 : b) = 42;  // OK, "*= 42" is brace-or-equal-initializer
    int z : 1 || new int { 0 }; // OK, brace-or-equal-initializer is absent
};
```

— end example]

10 A brace-or-equal-initializer shall appear only in the declaration of a data member. (For static data members, see 11.3.8.2; for non-static data members, see 11.9.2 and 9.3.1). 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 (11.8.3). A pure-specifier shall be used only in the declaration of a virtual function (11.6.2) 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-declarators; it shall not appear if the optional member-declarator-list is omitted.

14 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 (11.6.2).
15 The type of a non-static data member shall not be an incomplete type (6.7), an abstract class type (11.6.3),
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]
16 [Note: See 7.5.4 for restrictions on the use of non-static data members and non-static member functions.
— end note]
17 [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]
18 [Example: A simple example of a class definition is

```
struct tnode {
    char tword[20];
    int count;
    tnode* left;
    tnode* right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once
this definition has been given, the declaration

```
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]
19 Non-static data members of a (non-union) class with the same access control (11.8) 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 (11.8). 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 (11.6.2) and virtual base classes (11.6.1).
20 If T is the name of a class, then each of the following shall have a name different from T:

(20.1) — every static data member of class T;
(20.2) — every member function of class T [Note: This restriction does not apply to constructors, which do not
have names (11.3.4) — end note];
(20.3) — every member of class T that is itself a type;
(20.4) — every member template of class T;
(20.5) — every enumerator of every member of class T that is an unscoped enumerated type; and
(20.6) — every member of every anonymous union that is a member of class T.
21 In addition, if class T has a user-declared constructor (11.3.4), every non-static data member of class T shall
have a name different from T.
22 The common initial sequence of two standard-layout struct (11.1) 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 (9.11.11) or neither is, and either both entities are bit-fields with the same
width or neither is a bit-field. [Example:

```
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]
23 Two standard-layout struct (11.1) types are layout-compatible classes if their common initial sequence
comprises all members and bit-fields of both classes (6.7).
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 (6.7).

In a standard-layout union with an active member (11.4) 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 (9.1.7.1).

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 (6.7.2, 7.6.1.8). —end note**

11.3.1 Member functions

1 A member function may be defined (9.4) in its class definition, in which case it is an inline member function (9.1.6), or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. 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 (13.8) appearing outside of the class definition, a member function shall not be redeclared.

2 An inline member function (whether static or non-static) may also be defined outside of its class definition provided either its declaration in the class definition or its definition outside of the class definition declares the function as **inline** or **constexpr**. **Note:** Member functions of a class have the linkage of the name of the class. See 6.5. —end note

3 **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 6.2 and 9.1.6. —end note

4 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 (9.2.3.6) or in the member function body) is looked up as described in 6.4. —end note

**Example:**

```c
struct X {  
  typedef int T;  
  static T count;  
  void f(T 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

5 **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

6 Previously declared member functions may be mentioned in friend declarations.

7 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 9.2.3.5. For example,

```cpp
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 13.3. — end note]

11.3.2 Non-static member functions

A non-static member function may be called for an object of its class type, or for an object of a class derived (11.6) from its class type, using the class member access syntax (7.6.1.4, 12.3.1.1). A non-static member function may also be called directly using the function call syntax (7.6.1.2, 12.3.1.1) from within the body of a member function of its class or of a class derived from its class.

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.

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 (6.4) 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) (11.3.2.1) 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] Similarly during name lookup, when an unqualified-id (7.5.4.1) used in the definition of a member function for class \( X \) resolves to a static member, an enumerator or a nested type of class \( X \) or of a base class of \( X \), the unqualified-id is transformed into a qualified-id (7.5.4.2) in which the nested-name-specifier names the class of the member function. These transformations do not apply in the template definition context (13.7.2.1). [Example:

```cpp
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.\footnote{See, for example, <cstring> (21.5).}

A non-static member function may be declared \texttt{const}, \texttt{volatile}, or \texttt{const volatile}. These \texttt{cv-qualifiers} affect the type of the \texttt{this} pointer (11.3.2.1). They also affect the function type (9.2.3.5) of the member function; a member function declared \texttt{const} is a \texttt{const member function}, a member function declared \texttt{volatile} is a \texttt{volatile member function} and a member function declared \texttt{const volatile} is a \texttt{const volatile member function}.  

\footnotetext[106]{Example:}

\begin{verbatim}
struct X {
  void g() const;
  void h() const volatile;
};

X::g is a const member function and X::h is a const volatile member function. — end example\end{verbatim}

A non-static member function may be declared with a \texttt{ref-qualifier} (9.2.3.5); see 12.3.1.

A non-static member function may be declared virtual (11.6.2) or pure virtual (11.6.3).

11.3.2.1 The \texttt{this} pointer

In the body of a non-static (11.3.1) 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 of a class \texttt{X} is \texttt{X*}. If the member function is declared \texttt{const}, the type of \texttt{this} is \texttt{const X*}, if the member function is declared \texttt{volatile}, the type of \texttt{this} is \texttt{volatile X*}, and if the member function is declared \texttt{const volatile}, the type of \texttt{this} is \texttt{const volatile X*}.  

\footnotetext[1]{Note: Thus in a const member function, the object for which the function is called is accessed through a const access path. — end note}

\begin{verbatim}
struct s {
  int a;
  int f() const;
  int g() { return a++; }
  int h() const { return a++; } // error
};

int s::f() const { return a; }

The a++ in the body of s::h is ill-formed because it tries to modify (a part of) the object for which s::h() is called. This is not allowed in a const member function because this is a pointer to const; that is, *this has const type. — end example\end{verbatim}

Similarly, \texttt{volatile} semantics (9.1.7.1) apply in volatile member functions when accessing the object and its non-static data members.

A \texttt{cv-qualified} member function can be called on an object-expression (7.6.1.4) only if the object-expression is as \texttt{cv-qualified} or less-\texttt{cv-qualified} than the member function.  

\footnotetext[3]{Example:}

\begin{verbatim}
void k(s& x, const s& y) {
  x.f();
  x.g();
  y.f();
  y.g(); // error
}

The call y.g() is ill-formed because y is \texttt{const} and s::g() is a non-\texttt{const} member function, that is, s::g() is less-qualified than the object-expression y. — end example\end{verbatim}

\footnotetext[4]{Note: Constructors and destructors cannot be declared \texttt{const}, \texttt{volatile}, or \texttt{const volatile}. However, these functions can be invoked to create and destroy objects with \texttt{cv-qualified} types; see 11.3.4 and 11.3.6. — end note}

11.3.3 Special member functions

The default constructor (11.3.4.1), copy constructor, move constructor (11.3.4.2), copy assignment operator, move assignment operator (11.3.5), and destructor (11.3.6) are \texttt{special member functions}.  

\footnotetext[1]{Note: The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them. The implementation will implicitly define them if they are odr-used (6.2) or
needed for constant evaluation (7.7). — end note] An implicitly-declared special member function is declared at the closing } of the class-specifier. Programs shall not define implicitly-declared special member functions.

2 Programs may explicitly refer to implicitly-declared special member functions. [Example: A program may explicitly call or form a pointer to member to an implicitly-declared special member function.

```
struct A { };                     // implicitly declared A::operator=
struct B : A {
    B& operator=(const B &);
};
B& B::operator=(const B& s) {    // well-formed
    this->A::operator=(s);
    return *this;
}
```

— end example]

3 [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]

4 Special member functions obey the usual access rules (11.8). [Example: Declaring a constructor protected ensures that only derived classes and friends can create objects using it. — end example]

5 For a class, its non-static data members, its non-virtual direct base classes, and, if the class is not abstract (11.6.3), its virtual base classes are called its potentially constructed subobjects.

11.3.4 Constructors [class.ctor]

1 Constructors do not have names. In a declaration of a constructor, the declarator is a function declarator (9.2.3.5) of the form

\[
\text{ptr-declarator ( parameter-declaration-clause ) noexcept-specifier_{opt} attribute-specifier-seq_{opt}}
\]

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:

(1.1) — in a member-declaration that belongs to the member-specification of a class or class template but is not a friend declaration (11.8.3), the id-expression is the injected-class-name (Clause 11) of the immediately-enclosing entity or

(1.2) — in a declaration at namespace scope or in a friend declaration, the id-expression is a qualified-id that names a constructor (6.4.3.1).

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]

2 A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (7.6.1.3) 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) );
```

— end example] [Note: For initialization of objects of class type see 11.9. — end note]

3 An object created in this way is unnamed. [Note: 6.6.7 describes the lifetime of temporary objects. — end note] [Note: Explicit constructor calls do not yield lvalues, see 7.2.1. — end note]

4 [Note: Some language constructs have special semantics when used during construction; see 11.9.2 and 11.9.4. — end note]
A constructor can be invoked for a `const`, `volatile` or `const volatile` object. `const` and `volatile` semantics (9.1.7.1) are not applied on an object under construction. They come into effect when the constructor for the most derived object (6.6.2) ends.

A `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.

11.3.4.1 Default constructors

A `default constructor` for a class `X` is a constructor of class `X` for which each parameter that is not a function parameter pack has a default argument (including the case of a constructor with no parameters). If there is no user-declared constructor for class `X`, a non-explicit constructor having no parameters is implicitly declared as defaulted (9.4). An implicitly-declared default constructor is an inline public member of its class.

A defaulted default constructor for class `X` is defined as deleted if:

1. `X` is a union that has a variant member with a non-trivial default constructor and no variant member of `X` has a default member initializer,
2. `X` is a non-union class that has a variant member `M` with a non-trivial default constructor and no variant member of the anonymous union containing `M` has a default member initializer,
3. any non-static data member with no default member initializer (11.3) is of reference type,
4. any non-variant non-static data member of const-qualified type (or array thereof) with no `brace-or-equal-initializer` is not const-default-constructible (9.3).
5. `X` is a union and all of its variant members are of const-qualified type (or array thereof),
6. `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 `brace-or-equal-initializer`, has class type `M` (or array thereof) and either `M` has no default constructor or overload resolution (12.3) as applied to find `M`'s corresponding constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or
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 `trivial` if it is not user-provided and if:

1. its class has no virtual functions (11.6.2) and no virtual base classes (11.6.1), and
2. no non-static data member of its class has a default member initializer (11.3), 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 `non-trivial`.

A default constructor that is defaulted and not defined as deleted is `implicitly defined` when it is odr-used (6.2) to create an object of its class type (6.6.2), when it is needed for constant evaluation (7.7), or when it is explicitly defaulted after its first declaration. The implicitly-defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no `ctor-initializer` (11.9.2) and an empty `compound-statement`. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor (9.1.5), the implicitly-defined default constructor is `constexpr`. Before the defaulted default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [Note: An implicitly-declared default constructor has an exception specification (14.4). An explicitly-defaulted definition might have an implicit exception specification, see 9.4. — end note]

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (6.6.5.1, 6.6.5.2, 6.6.5.3) defined without an initializer (9.3), are called to create class objects of dynamic storage duration (6.6.5.4) created by a `new-expression` in which the `new-initializer` is omitted (7.6.2.5), 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 (11.8).
[Note: 11.9.2 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]

11.3.4.2 Copy/move constructors

A non-template constructor for class \( X \) is a copy constructor if its first parameter is of type \( X\& \), \( \text{const } X\& \), \( \text{volatile } X\& \) or \( \text{const volatile } X\& \), and either there are no other parameters or else all other parameters have default arguments (9.2.3.6). [Example: \( X::X(\text{const } X\&) \) and \( X::X(X\&,\text{int}=1) \) 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]
```

A non-template constructor for class \( X \) is a move constructor if its first parameter is of type \( X&& \), \( \text{const } X&& \), \( \text{volatile } X&& \), or \( \text{const volatile } X&& \), and either there are no other parameters or else all other parameters have default arguments (9.2.3.6). [Example: \( Y::Y(Y&&) \) is a move constructor.]

```cpp
struct Y {
    Y(const Y&);
    Y(Y&&); // OK
    X a; // OK, but possibly not sensible
};
extern Y f(int);
Y d(f(1)); // calls Y(Y&)
Y e = d; // calls Y(const Y&)
—end example]
```

[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]
```

[Note: If a class \( X \) only has a copy constructor with a parameter of type \( X\& \), an initializer of type \( \text{const } X \) or \( \text{volatile } X \) cannot initialize an object of type (possibly cv-qualified) \( X \). [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]
```

A declaration of a constructor for a class \( X \) is ill-formed if its first parameter is of type (optionally cv-qualified) \( X \) 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);
    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
}
```
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 (9.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor (D.5).

The implicitly-declared copy constructor for a class \( X \) will have the form
\[
X::X(const X&) \tag{6}
\]
if each potentially constructed subobject of a class type \( M \) (or array thereof) has a copy constructor whose first parameter is of type \( \text{const } M & \) or \( \text{const volatile } M & \).\(^{107}\) Otherwise, the implicitly-declared copy constructor will have the form
\[
X::X(X&) \tag{7}
\]

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

- \( X \) does not have a user-declared copy constructor,
- \( X \) does not have a user-declared copy assignment operator,
- \( X \) does not have a user-declared move assignment operator, and
- \( X \) 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]

The implicitly-declared move constructor for class \( X \) will have the form
\[
X::X(X&&) \tag{8}
\]

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 (9.4.3) if \( X \) has:

- a potentially constructed subobject type \( M \) (or array thereof) that cannot be copied/moved because overload resolution (12.3), as applied to find \( M \)'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
- a variant member whose corresponding constructor as selected by overload resolution is non-trivial,
- any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,
- for the copy constructor, a non-static data member of rvalue reference type.

A defaulted move constructor that is defined as deleted is ignored by overload resolution (12.3, 12.4). [Note: A deleted move constructor would otherwise interfere with initialization from an rvalue which can use the copy constructor instead. — end note]

A copy/move constructor for class \( X \) is trivial if it is not user-provided and if:

- class \( X \) has no virtual functions (11.6.2) and no virtual base classes (11.6.1), and
- the constructor selected to copy/move each direct base class subobject is trivial, and
- for each non-static data member of \( X \) that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;

otherwise the copy/move constructor is *non-trivial*.

A copy/move constructor that is defaulted and not defined as deleted is *implicitly defined* when it is odr-used (6.2), 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 (6.2, 6.6.7). — end note] If the implicitly-defined constructor would satisfy the requirements of a constexpr constructor (9.1.5), the implicitly-defined constructor is \text{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 shall have been implicitly defined. [Note: An implicitly-declared copy/move constructor has an implied exception specification (14.4). — end note]

\(^{107}\) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a volatile lvalue; see C.1.6.
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 11.9.2. — end note] The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 11.9.2). 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 \( \text{T&&} \), it is direct-initialized with \( \text{static_cast<\text{T&&}}(x.m) \);
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 11.9.2).

The implicitly-defined copy/move constructor for a union \( X \) copies the object representation (6.7) of \( X \).

### 11.3.5 Copy/move assignment operator

A user-declared copy assignment operator \( X::\text{operator=} \) is a non-static non-template member function of class \( X \) with exactly one parameter of type \( X, X&, \text{const X&}, \text{volatile X&}, \text{or const volatile X&}. \)[Note: An overloaded assignment operator must be declared to have only one parameter; see 12.5.3. — 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 \( \text{const X} \) cannot be assigned to an object of type \( X \). [Example:

```cpp
struct X {
    X();
    X& operator=(X&);
};
const X cx;
X x;
void f() {
    x = cx; // error: X::operator=(X&) cannot assign cx into x
}
end example] — end note]

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 (9.4). The latter case is deprecated if the class has a user-declared copy constructor or a user-declared destructor (D.5). 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 \( \text{const B&}, \text{const volatile B&}, \text{or B}, \text{and} \)
2. for all the non-static data members of \( X \) that are of a class type \( M \) (or array thereof), each such class type has a copy assignment operator whose parameter is of type \( \text{const M&}, \text{const volatile M&}, \text{or M}. \)[109]

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 \( X&&, \text{const X&&}, \text{volatile X&&}, \text{or const volatile X&&}. \)[Note: An overloaded assignment operator must be declared to have only one parameter; see 12.5.3. — end note] [Note: More than one form of move assignment operator may be declared for a class. — end note]

[108] Because a template assignment operator or an assignment operator taking an rvalue reference parameter is never a copy assignment operator, the presence of such an assignment operator does not suppress the implicit declaration of a copy assignment operator. Such assignment operators participate in overload resolution with other assignment operators, including copy assignment operators, and, if selected, will be used to assign an object.

[109] This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a \text{volatile} lvalue; see C.1.6.
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

```cpp
struct S {
    int a;
    S& operator=(const S&) = 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.

```cpp
struct S {
    int a;
    S& operator=(const S&) = default;
    S& operator=(S&&) = default;
}; // end example
```

The implicitly-declared move assignment operator for a class $X$ will have the form

$$X& X::operator=(X&&);$$

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 `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 (12.3), 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.

A defaulted move assignment operator that is defined as deleted is ignored by overload resolution (12.3, 12.4).

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 (12.5.3). A `using-declaration` (9.8) that brings in from a base class an assignment operator with a parameter type that could be that of a copy/move assignment operator for the derived class is not considered an explicit declaration of such an operator and does not suppress the implicit declaration of the derived class operator; the operator introduced by the `using-declaration` is hidden by the implicitly-declared operator in the derived class.

A copy/move assignment operator for class $X$ is trivial if it is not user-provided and if:

1. class $X$ has no virtual functions (11.6.2) and no virtual base classes (11.6.1), and
2. the assignment operator selected to copy/move each direct base class subobject is trivial, and
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 (6.2) (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
© ISO/IEC N4810

11 Before the defaulted copy/move assignment operator for a class is implicitly defined, all non-user-provided
copy/move assignment operators for its direct base classes and its non-static data members shall have been
implicitly defined. [Note: An implicitly-declared copy/move assignment operator has an implied exception
specification (14.4). — end note]

12 The implicitly-defined copy/move assignment operator for a non-union class \( X \) performs memberwise copy-
move assignment of its subobjects. The direct base classes of \( X \) are assigned first, in the order of their
declaration in the \textit{base-specifier-list}, and then the immediate non-static data members of \( X \) are assigned, in
the order in which they were declared in the class definition. Let \( x \) be either the parameter of the function
or, for the move operator, an xvalue referring to the parameter. Each subobject is assigned in the manner
appropriate to its type:

\begin{enumerate}
\item if the subobject is of class type, as if by a call to \texttt{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);
\item if the subobject is an array, each element is assigned, in the manner appropriate to the element type;
\item if the subobject is of scalar type, the built-in assignment operator is used.
\end{enumerate}

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the
implicitly-defined copy/move assignment operator. [Example:

\begin{verbatim}
struct V {
};
struct A : virtual V {
};
struct B : virtual V {
};
struct C : B, A {
};
\end{verbatim}

It is unspecified whether the virtual base class subobject \( V \) is assigned twice by the implicitly-defined
copy/move assignment operator for \( C \). — end example]

13 The implicitly-defined copy assignment operator for a union \( X \) copies the object representation (6.7) of \( X \).

\section*{11.3.6 Destructors \textit{[class.dtor]}}
\begin{enumerate}
\item In a declaration of a destructor, the \textit{declarator} is a function declarator (9.2.3.5) of the form
\begin{verbatim}
ptr-declarator ( parameter-declaration-clause ) noexcept-specifier_opt attribute-specifier-seq_opt
\end{verbatim}
where the \textit{ptr-declarator} consists solely of an \textit{id-expression}, an optional \textit{attribute-specifier-seq}, and optional
surrounding parentheses, and the \textit{id-expression} has one of the following forms:
\begin{enumerate}
\item in a \textit{member-declaration} that belongs to the \textit{member-specification} of a class or class template but is not a
friend declaration (11.8.3), the \textit{id-expression} is \texttt{~class-name} and the \textit{class-name} is the injected-class-
name (Clause 11) of the immediately-enclosing entity or
\item in a declaration at namespace scope or in a friend declaration, the \textit{id-expression} is \texttt{nested-name-specifier
\texttt{~class-name}} and the \textit{class-name} names the same class as the \textit{nested-name-specifier}.
\end{enumerate}
A destructor shall take no arguments (9.2.3.5). Each \texttt{decl-specifier} of the \texttt{decl-specifier-seq} of a destructor
declaration (if any) shall be \texttt{friend}, \texttt{inline}, or \texttt{virtual}.
\item The address of a destructor shall not be taken. A destructor can be invoked for a \texttt{const}, \texttt{volatile} or \texttt{const}
\texttt{volatile} object. \texttt{const} and \texttt{volatile} semantics (9.1.7.1) are not applied on an object under destruction.
They stop being in effect when the destructor for the most derived object (6.6.2) starts.
\item [Note: A declaration of a destructor that does not have a \texttt{noexcept-specifier} has the same exception specification
as if it had been implicitly declared (14.4). — end note]
\item If a class has no user-declared destructor, a destructor is implicitly declared as defaulted (9.4). An implicitly-
declared destructor is an inline public member of its class.
\item A defaulted destructor for a class \( X \) is defined as deleted if:
\begin{enumerate}
\item \( X \) 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 \textit{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 \textit{non-trivial}.

A destructor that is defaulted and not defined as deleted is \textit{implicitly defined} when it is odr-used (6.2) or when it is explicitly defaulted after its first declaration.

Before the defaulted destructor for a class is implicitly defined, all the non-user-provided destructors for its base classes and its non-static data members shall have been implicitly defined.

After executing the body of the destructor and destroying any automatic objects 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 (11.9.2), 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 11.9.2). A \texttt{return} statement (8.6.3) 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 11.9).

A destructor can be declared \textit{virtual} (11.6.2) or pure \textit{virtual} (11.6.3); if 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 11.9.4. — end note]

A destructor is invoked implicitly

— for a constructed object with static storage duration (6.6.5.1) at program termination (6.8.3.4),

— for a constructed object with thread storage duration (6.6.5.2) at thread exit,

— for a constructed object with automatic storage duration (6.6.5.3) when the block in which an object is created exits (8.7),

— for a constructed temporary object when its lifetime ends (7.3.4, 6.6.7).

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 \texttt{delete-expression} (7.6.2.6) for a constructed object allocated by a \texttt{new-expression} (7.6.2.5); the context of the invocation is the \texttt{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 \textit{potentially invoked} if it is invoked or as specified in 7.6.2.5, 9.3.1, 11.9.2, and 14.1. A program is ill-formed if a destructor that is potentially invoked is deleted or not accessible from the context of the invocation.

At the point of definition of a virtual destructor (including an implicit definition (11.3.6)), the non-array deallocation function is determined as if for the expression \texttt{delete this} appearing in a non-virtual destructor of the destructor’s class (see 7.6.2.6). If the lookup fails or if the deallocation function has a deleted definition (9.4), the program is ill-formed. [Note: This assures that a deallocation function corresponding to the dynamic type of an object is available for the \texttt{delete-expression} (11.11). — end note]

In an explicit destructor call, the destructor is specified by a \texttt{~} followed by a \texttt{type-name} or \texttt{decltype-specifier} that denotes the destructor’s class type. The invocation of a destructor is subject to the usual rules for member functions (11.3.1); 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 \texttt{delete} on a null pointer does not call the destructor; see 7.6.2.6. — end note] [Example:
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 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

15 [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,]

void* operator new(std::size_t, void* p) { return p; }
struct X {
    X(int);
    ~X();
};
void f(X* p);

void g() {
    // rare, specialized use:
    char* buf = new char[sizeof(X)];
    X* p = new(buf) X(222); // use buf[] and initialize
    f(p);
    p->X::~X(); // cleanup
}

—end note

16 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 (6.6.3). [Example: If the destructor for an automatic object 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]

17 [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,]

typedef int I;
I* p;
p->I::~I();

—end note

18 A destructor shall not be a coroutine.

11.3.7 Conversions [class.conv]

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 (9.3), and for explicit type conversions (7.6.3, 7.6.1.8).

User-defined conversions are applied only where they are unambiguous (11.7, 11.3.7.2). Conversions obey the access control rules (11.8). Access control is applied after ambiguity resolution (6.4).
3 [Note: See 12.3 for a discussion of the use of conversions in function calls as well as examples below. — end note]

4 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value. [Example:

```c
struct X {
    operator int();
};

struct Y {
    operator X();
};

Y a;
int b = a; // error, a.operator X().operator int() not tried
int c = X(a); // OK: a.operator X().operator int()
```
—end example]

5 User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same type. Function overload resolution (12.3.3) selects the best conversion function to perform the conversion. [Example:

```c
struct X {
    operator int();
};

struct Y : X {
    operator char();
};

void f(Y& a) {
    if (a) {
        // ill-formed: X::operator int() or Y::operator char()
    }
}
```
—end example]

### 11.3.7.1 Conversion by constructor [class.conv.ctor]

1 A constructor that is not explicit (9.1.2) 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:

```c
struct X {
    X(int);
    X(const char*, int =0);
    X(int, int);
};

void f(X arg) {
    X a = 1; // a = X(1)
    X b = "Jessie"; // b = X("Jessie",0)
    a = 2; // a = X(2)
    f(3); // f(X(3))
    f({1, 2}); // f(X(1,2))
}
```
—end example]

2 [Note: An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (9.3) or where casts (7.6.1.8, 7.6.3) are explicitly used; see also 12.3.1.4. A default constructor may be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (9.3). [Example:

```c
struct Z {
    explicit Z();
    explicit Z(int);
```
explicit Z(int, int);
};

Z a;       // OK: default-initialization performed
Z b();    // OK: direct initialization syntax used
Z c = {}; // error: copy-list-initialization
Z a1 = 1;  // error: no implicit conversion
Z a3 = Z(1); // OK: direct initialization syntax used
Z a2(1);  // OK: direct initialization syntax used
Z* p = new Z(1); // OK: direct initialization syntax used
Z a4 = (Z)1; // OK: explicit cast used
Z a5 = static_cast<Z>(1); // OK: explicit cast used
Z a6 = { 3, 4 }; // error: no implicit conversion

—end example — end note

A non-explicit copy/move constructor (11.3.4.2) is a converting constructor. [Note: An implicitly-declared copy/move constructor is not an explicit constructor; it may be called for implicit type conversions. — end note]

11.3.7.2 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 (9.2.3.5) 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 (possibly cv-qualified) void.110 [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]

A conversion function may be explicit (9.1.2), in which case it is only considered as a user-defined conversion for direct-initialization (9.3). 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;      // ill-formed: 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:

```
&ac.operator int*1;  // syntax error;
    // parsed as: k(ac.operator int *)i
    // not as: k(ac.operator int)*i
```

The * is the pointer declarator and not the multiplication operator. — end example] This rule also prevents ambiguities for attributes. [Example:

```
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.

6 A conversion function template shall not have a deduced return type (9.1.7.5). [Example:

```
struct S {
    operator auto() const { return 10; }   // OK
    template<class T>
    operator auto() const { return 1.2; }  // error: conversion function template
};
— end example]

11.3.8 Static members [class.static]

1 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:

```
struct process {
    static void reschedule();
};
process& g();

void f() {
    process::reschedule();  // OK: no object necessary
g().reschedule();        // g() is called
}
— end example]

2 A static member may be referred to directly in the scope of its class or in the scope of a class derived (11.6) 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:

```
int g();
struct X {
    static int g();
};
struct Y : X {
    static int i;
};
int Y::i = g();          // equivalent to Y::g();
```
If an unqualified-id (7.5.4.1) is used in the definition of a static member following the member’s declarator-id, and name lookup (6.4.1) finds that the unqualified-id refers to a static member, enumerator, or nested type of the member’s class (or of a base class of the member’s class), the unqualified-id is transformed into a qualified-id expression in which the nested-name-specifier names the class scope from which the member is referenced.  

[Note: See 7.5.4 for restrictions on the use of non-static data members and non-static member functions. — end note]

Static members obey the usual class member access rules (11.8). 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]

### 11.3.8.1 Static member functions

[Note: The rules described in 11.3.1 apply to static member functions. — end note]

A static member function does not have a this pointer (11.3.2.1).  

[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 (12.1). A static member function shall not be declared const, volatile, or const volatile. — end note]

### 11.3.8.2 Static data members

A static data member is not part of the subobjects of a class. If a static data member is declared thread-local there is one copy of the member per thread. If a static data member is not declared thread_local there is one copy of the data member that is shared by all the objects of the class.  

The declaration of a non-inline static data member in its class definition is not a definition and may be of an incomplete type other than cv void. The definition for a static data member that is not defined inline in the class definition shall appear in a namespace scope enclosing the member’s class definition. In the definition at namespace scope, the name of the static data member shall be qualified by its class name using the :: operator. The initializer expression in the definition of a static data member is in the scope of its class (6.3.7).  

[Example:

```cpp
class process {
    static process* run_chain;
    static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The static data member run_chain of class process is defined in global scope; the notation process::run-chain specifies that the member run_chain is a member of class process and in the scope of class process. In the static data member definition, the initializer expression refers to the static data member running of class process. — end example]

[Note: Once the static data member has been defined, it exists even if no objects of its class have been created. [Example: In the example above, run_chain and running exist even if no objects of class process are created by the program. — end example] — end note]

If a non-volatile non-inline const static data member is of integral or enumeration type, its declaration in the class definition can specify a brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression (7.7). The member shall still be defined in a namespace scope if it is odr-used (6.2) in the program and the namespace scope definition shall not contain an initializer. An inline static data member may be defined in the class definition and may specify a brace-or-equal-initializer. If the member is declared with the constexpr specifier, it may be redeclared in namespace scope with no initializer (this usage is deprecated; see D.4). Declarations of other static data members shall not specify a brace-or-equal-initializer.  

[Note: There shall be exactly one definition of a static data member that is odr-used (6.2) in a program; no diagnostic is required. — end note] Unnamed classes and classes contained directly or indirectly within unnamed classes shall not contain static data members.

[Note: Static data members of a class in namespace scope have the linkage of the name of the class (6.5). A local class cannot have static data members (11.5). — end note]
Static data members are initialized and destroyed exactly like non-local variables (6.8.3.2, 6.8.3.3, 6.8.3.4).

A static data member shall not be mutable (9.1.1).

11.3.9 Bit-fields

A member-declarator of the form

```plaintext
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 (6.7). Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [Note: Bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. —end note]

A declaration for a bit-field that omits the identifier declares an unnamed bit-field. Unnamed bit-fields are not members and cannot be initialized. An unnamed bit-field shall not be declared with a cv-qualified type. [Note: An unnamed bit-field is useful for padding to conform to externally-imposed layouts. —end note]

As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the width be zero.

The address-of operator & shall not be applied to a bit-field, so there are no pointers to bit-fields. A non-const reference shall not be bound to a bit-field (9.3.3). [Note: If the initializer for a reference of type const T& is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly. See 9.3.3. —end note]

If a value of integral type (other than bool) is stored into a bit-field of width N and the value would be representable in a hypothetical signed or unsigned integer type with width N and the same signedness as the bit-field’s type, the original value and the value of the bit-field compare equal. If the value true or false is stored into a bit-field of type bool of any size (including a one bit bit-field), the original bool value and the value of the bit-field compare equal. If a value of an enumeration type is stored into a bit-field of the same type and the width is large enough to hold all the values of that enumeration type (9.6), the original value and the value of the bit-field compare equal. [Example:

```plaintext
genum BOOL { FALSE=0, TRUE=1 }; 
struct A { 
 BOOL b:1; 
}; 
a; 
void f() {
 a.b = TRUE; 
 if (a.b == TRUE) // yields true
   { /* ... */ }
}
—end example]

11.3.10 Nested class declarations

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:

```plaintext
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 (11.8.3) 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 (11.3.8), but it has no special access rights to members of an enclosing class.

11.3.11 Nested type names [class.nested.type]

1 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 { /* ... */ };
    I a;
};

I b;  // error
Y c;   // error
X::Y d; // OK
X::I e; // OK
—end example]
```
11.4 Unions

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 (6.6.3). 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 (11.3), 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 11.3. — 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 (6.7.2, 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 (11.6.2) 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 (11.3), if any non-static data member of a union has a non-trivial default constructor (11.3.4.1), copy constructor, move constructor (11.3.4.2), copy assignment operator, move assignment operator (11.3.5), or destructor (11.3.6), the corresponding member function of the union must be user-provided or it will be implicitly deleted (9.4.3) for the union. — end note]

[Example: Consider the following union:

```cpp
union U {
    int i;
    float f;
    std::string s;
};
```

Since `std::string` (21.3) declares non-trivial versions of all of the special member functions, `U` will have an implicitly deleted default constructor, copy/move constructor, copy/move assignment operator, and destructor. To use `U`, some or all of these member functions must be user-provided. — end example]

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 (11.3.5), for each element `X` of `S(E1)`, if modification of `X` would have undefined behavior under 6.6.3, 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 (6.6.3). — end note]  

```cpp
union A { int x; int y[4]; };  
struct B { A a; };  
union C { B b; int k; };  
int f() {  
    C c;  // does not start lifetime of any union member  
    c.b.a.y[3] = 4;  // OK: S(c.b.a.y[3]) contains c.b and c.b.a.y  
    // creates objects to hold union members c.b and c.b.a.y  
    return c.b.a.y[3];  // OK: c.b.a.y refers to newly created object (see 6.6.3)  
}  
struct X { const int a; int b; };  
union Y { X x; int k; };  
```
void g() {
    Y y = { { 1, 2 } }; // OK, y.x is active union member (11.3)
    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
    y.x.b = n; // undefined behavior: y.x.b modified outside its lifetime,
               // so union member y.x's lifetime does not implicitly start
}

[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:

    u.m.~M();
    new (&u.n) N;

— end example]

11.4.1 Anonymous unions [class.union.anon]

A union of the form

    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. [Note: Nested types, anonymous unions, and functions cannot be declared within an anonymous union. — end note] The names of the members of an anonymous union shall be distinct from the names of any other entity in the scope in which the anonymous union is declared. For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared. [Example:

    void f() {
        union { int a; const char* p; };  // error
        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 (11.8). An anonymous union shall not have member functions.

A union for which objects, pointers, or references are declared is not an anonymous union. [Example:

    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 9.3.1. — 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:
union U {
    int x = 0;
    union {
        int k;
    };
    union {
        int z;
        int y = 1;  // error: initialization for second variant member of U
    };
};
—end example]

11.5 Local class declarations

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 (6.2) a local entity from an enclosing scope. — end note] [Example:

    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 structured binding y
            int p() { return y; }   // error: odr-use of non-odr-usable variable
        }
    }
    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 (11.8). 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 A local class shall not have static data members.

11.6 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-seqopt class-or-decltype
    attribute-specifier-seqopt virtual access-specifier_opt class-or-decltype
    attribute-specifier-seqopt access-specifier virtual_opt class-or-decltype

§ 11.6
class-or-decltype:
  nested-name-specifier, type-name
  nested-name-specifier template simple-template-id
dectype-specifier

access-specifier:
  private
  protected
  public

The optional attribute-specifier-seq appertains to the base-specifier.

A class-or-decltype shall denote a (possibly cv-qualified) class type that is not an incompletely defined class (11.3); 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 (6.3.10). 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 11.8 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 9.8. 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 (11.7). [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 11.8.2. 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 (9.3.3). — end note]

The base-specifier-list specifies the type of the base class subobjects contained in an object of the derived class type. [Example:

```cpp
struct Base {
    int a, b, c;
};

struct Derived : Base {
    int b;
};

struct Derived2 : Derived {
    int c;
};
```

Here, an object of class Derived2 will have a subobject of class Derived which in turn will have a subobject of class Base. — end example]

A base-specifier followed by an ellipsis is a pack expansion (13.6.3).

The order in which the base class subobjects are allocated in the most derived object (6.6.2) 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”. An arrow need not have a physical representation in memory. A DAG of subobjects is often referred to as a “subobject lattice”.

```
   Base
   ↑
  Derived1
  ↑
Derived2
```

Figure 2 — Directed acyclic graph
6 [Note: Initialization of objects representing base classes can be specified in constructors; see 11.9.2. — end note]

7 [Note: A base class subobject might have a layout (6.6.5) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (11.9.4) 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]

11.6.1 Multiple base classes

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 (11.9.2), cleanup (11.3.6), and storage layout (11.3.1, 11.8.1). — 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 { /* ... */ };
// ill-formed
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ };
// well-formed
class D : public A, public L { void f(); /* ... */ };
// well-formed
— end example]

4 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 (6.6.2) 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.

5 [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.

```cpp
Figure 3 — Non-virtual base

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

§ 11.6.1
Without the A:: or B:: qualifiers, the definition of C::f above would be ill-formed because of ambiguity (11.7).

—end note

6 [Note: In contrast, consider the case with a virtual base class:

```c++
class V { /* ... */ }
class A : virtual public V { /* ... */ }
class B : virtual public V { /* ... */ }
class C : public A, public B { /* ... */ }
```

![Figure 4 — Virtual base](image)

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]

7 [Note: A class can have both virtual and non-virtual base classes of a given type.

```c++
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](image)

—end note]

11.6.2 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).[111] [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.[112]

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 (9.2.3.5), cv-qualification, and

---

111) The use of the virtual specifier in the declaration of an overriding function is valid but redundant (has empty semantics).
112) If all virtual functions are immediate functions, the class is still polymorphic even though its internal representation might not otherwise require any additions for that polymorphic behavior.
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 (6.6.2) 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:

```cpp
struct A {
    virtual void f();
};
struct B : virtual A {
    virtual void f();
};
struct C : B, virtual A {
    using A::f;
};

void foo() {
    C c;
    c.f(); // calls B::f, the final overrider
    c.C::f(); // calls A::f because of the using-declaration
}
```
—end example]

[Example:
```c
struct A { virtual void f();};
struct B : A { void f();};
struct D : B, C { }; // OK: A::f and C::f are the final overriders
    // for the B and C subobjects, respectively
```—end example]

3 [Note: 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:

113) A function with the same name but a different parameter list (Clause 12) as a virtual function is not necessarily virtual and does not override. Access control (11.8) is not considered in determining overriding.
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 (9.2). [Example:

struct A {
    virtual void f() requires true; // error: virtual function cannot be constrained (13.4.2)
};

— end example]

Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 11.3.6 and 11.11.

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:

(8.1) both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes
(8.2) 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
(8.3) both pointers or references have the same cv-qualification and the class type in the return type of D::f has the same cv-qualification as or less cv-qualification than the class type in the return type of B::f.

If the class type in the covariant return type of D::f differs from that of B::f, the class type in the return type of D::f shall be complete at the point of declaration of D::f or shall be the class type D. When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (7.6.1.2). [Example:

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;

114) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.
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();  // ill-formed: 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 virtual specifier implies membership, so a virtual function cannot be a non-member (9.1.2)
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 (11.8.3) in another class. — end note]

A virtual function declared in a class shall be defined, or declared pure (11.6.3) in that class, or both; no
diagnostic is required (6.2).

[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* b1p = &d;
    A* ap = b1p;
    D* dp = &d;
    ap->f();  // calls D::B1::f
    dp->f();  // ill-formed: 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]

[Example: The following example shows a function that does not have a unique final overrider:

```cpp
struct A {
    virtual void f();
};

struct VB1 : virtual A {  // note virtual derivation
    void f();
};
```

§ 11.6.2
struct VB2 : virtual A {
    void f();
};

struct Error : VB1, VB2 {       // ill-formed
};

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* vb1ap = new Da;
    vb1ap->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(); };  
class D : public B { public: void f(); };  

void D::f() { /* ... */ B::f(); }

Here, the function call in D::f really does call B::f and not D::f. —end example]

17 A function with a deleted definition (9.4) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

18 A constexpr virtual function shall not override a virtual function that is not constexpr. A constexpr virtual function shall not be overridden by a virtual function that is not constexpr.

19 If an overriding function specifies contract conditions (9.11.4), it shall specify the same list of contract conditions as its overridden functions; no diagnostic is required if corresponding conditions will always evaluate to the same value. Otherwise, it is considered to have the list of contract conditions from one of its overridden functions; the names in the contract conditions are bound, and the semantic constraints are checked, at the point where the contract conditions appear. Given a virtual function f with a contract condition that odr-uses *this (6.2), the class of which f is a direct member shall be be an unambiguous and accessible base class of any class in which f is overridden. If a function overrides more than one function, all of the overridden functions shall have the same list of contract conditions (9.11.4); no diagnostic is required if corresponding conditions will always evaluate to the same value. [Example:

struct A {
    virtual void g() [[expects: x == 0]];  
    int x = 42;
};

int x = 42;
struct B {
    virtual void g() [[expects: x == 0]];
}

struct C : A, B { 
    virtual void g();               // error: preconditions of overridden functions are not the same
};

§ 11.6.2 276
11.6.3 Abstract classes

[Note: The abstract class mechanism supports the notion of a general concept, such as a shape, of which only more concrete variants, such as circle and square, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations. —end note]

A virtual function is specified as a pure virtual function by using a pure-specifier (11.3) in the function declaration in the class definition. [Note: Such a function might be inherited; see below. —end note] A class is an abstract class if it has at least one pure virtual function. [Note: An abstract class can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it (6.1, 11.3). —end note] A pure virtual function need be defined only if called with, or as if with (11.3.6), the qualified-id syntax (7.5.4.2). [Example:

```cpp
class point { /* ... */};
class shape { // abstract class
    point center;
    public:
        point where() { return center; }
        void move(point p) { center=p; draw(); }
        virtual void rotate(int) = 0; // pure virtual
        virtual void draw() = 0; // pure virtual
    }

    // end example] [Note: A function declaration cannot provide both a pure-specifier and a definition —end note] [Example:

```cpp
struct C {
    virtual void f() = 0 { }; // ill-formed
};

    // end example]

[Note: An abstract class type cannot be used as a parameter or return type of a function being defined (9.2.3.5) or called (7.6.1.2), except as specified in 9.1.7.2. 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]

A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [Example:

```cpp
class ab_circle : public shape {
    int radius;
    public:
        void rotate(int) { }
        // ab_circle::draw() is a pure virtual
    }

    // end example]

Since shape::draw() is a pure virtual function ab_circle::draw() is a pure virtual by default. The alternative declaration,

```cpp
class circle : public shape {
    int radius;
    public:
        void rotate(int) { }
        void draw(); // a definition is required somewhere
    }

    // end example]

would make class circle non-abstract and a definition of circle::draw() must be provided. —end example]

[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]

Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (11.6.2) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.
11.7 Member name lookup

Member name lookup determines the meaning of a name (id-expression) in a class scope (6.3.7). 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 (6.4, 11.8).

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 (9.8), and type declarations (including injected-class-names) are replaced by the types they designate. \( S(f,C) \) is calculated as follows:

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 (6.4.4) or base-specifier (11.6), for instance, ignores all non-type declarations, while looking up a name in a nested-name-specifier (6.4.3) ignores function, variable, and enumerator declarations. As another example, looking up a name in a using-declaration (9.8) 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.

Otherwise (i.e., \( C \) does not contain a declaration of \( f \) or the resulting declaration set is empty), \( S(f,C) \) is initially empty. If \( C \) has base classes, calculate the lookup set for \( f \) in each direct base class subobject \( B_i \), and merge each such lookup set \( S(f,B_i) \) in turn into \( S(f,C) \).

The following steps define the result of merging lookup set \( S(f,B_i) \) into the intermediate \( S(f,C) \):

- 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 \( S(f,B_i) \).

- 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.

- Otherwise, the new \( S(f,C) \) is a lookup set with the shared set of declarations and the union of the subobject sets.

The result of name lookup for \( f \) in \( C \) is the declaration set of \( S(f,C) \). If it is an invalid set, the program is ill-formed. [Example:

```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]

If the name of an overloaded function is unambiguously found, overload resolution (12.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [Example:

```c
struct A {
    int f();
};
struct B {
    int f();
};
```
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:

```cpp
struct V {
    int v;
};
struct A {
    int a;
    static int s;
    enum { e };
};
struct B : A, virtual V { }
struct C : A, virtual V { }
struct D : B, C { }

void f(D* pd) {
    pd->v++;       // OK: only one v (virtual)
    pd->s++;       // OK: only one s (static)
    int i = pd->e; // OK: only one e (enumerator)
    pd->a++;       // error, ambiguous: two a in D
}
```

—end example]

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:

```cpp
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(); };

void D::glorp() {
    x++;       // OK: B::x hides V::x
    f();       // OK: B::f() hides V::f()
    y++;       // error: B::y and C's W::y
    g();       // error: B::g() and C's W::g()
}
```

Figure 6 — Name lookup

The names declared in V and the left-hand instance of W are hidden by those in B, but the names declared in the right-hand instance of W are not hidden at all.

void D::glorp()
```
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:

```c
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: C’s A or B’s A?
    V* pv = &d;  // OK: only one V subobject
}
```
—end example]

[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, 11.8.2). — end note] [Example:

```c
struct B1 {
    void f();
    static void f(int);
    int i;
};
struct B2 {
    void f(double);
};
struct I1: B1 { }
struct I2: B1 { }
struct D: I1, I2, B2 {
    using B1::f;
    using B2::f;
    void g() {
        f();  // Ambiguous conversion of this
        f(0);  // Unambiguous (static)
        f(0.0);  // Unambiguous (only one B2)
        int B1::* mpB1 = &D::i;
        // Unambiguous
        int D::* mpD = &D::i;  // Ambiguous conversion
    }
};
```
—end example]

## 11.8 Member access control

A member of a class can be

(1.1) — private; that is, its name can be used only by members and friends of the class in which it is declared.

(1.2) — protected; that is, its name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends (see 11.8.4).

(1.3) — public; that is, its name can be used anywhere without access restriction.

A member of a class can also access all the names to which the class has access. A local class of a member function may access the same names that the member function itself may access.\(^{115}\)

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:

```c
class X {
    int a;  // X::a is private by default
};
```

\(^{115}\) Access permissions are thus transitive and cumulative to nested and local classes.
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 (11.8.3) and using-declarations (9.8). — 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 (6.3.10). 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 11.8 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 { }

    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 (9.2.3.6) 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 13.8.1.

The names in a default template-argument (13.1) have their access checked in the context in which they
appear rather than at any points of use of the default template-argument. [Example:

```cpp
class B { }

template <class T> class C {
  protected:
  
  typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U { }

D <C<B> > d;  // access error, C::TT is protected
```
—end example]

11.8.1 Access specifiers

Member declarations can be labeled by an access-specifier (11.6):

```
access-specifier : member-specification
```

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:

```cpp
class X {
  int a;  // X::a is private by default: class used
  public:
  int b;  // X::b is public
  int c;  // X::c is public
};
```
—end example]

Any number of access specifiers is allowed and no particular order is required. [Example:

```cpp
struct S {
  int a;  // S::a is public by default: struct used
  protected:
  int b;  // S::b is protected
  int c;  // S::c is private
  public:
  int d;  // S::d is public
};
```
—end example]

[Note: The effect of access control on the order of allocation of data members is described in 11.3. — end note]

When a member is redeclared within its class definition, the access specified at its redeclaration shall be the
same as at its initial declaration. [Example:

```cpp
struct S {
  class A;
  enum E : int;
  private:
    class A { };  // error: cannot change access
    enum E : int { a0 };  // 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]
11.8.2 Accessibility of base classes and base class members [class.access.base]

1 If a class is declared to be a base class (11.6) 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:

```cpp
class B { /* ... */ }; class D1 : private B { /* ... */ }; class D2 : public B { /* ... */ }; class D3 : B { /* ... */ }; // B private by default struct D4 : public B { /* ... */ }; struct D5 : private B { /* ... */ }; struct D6 : B { /* ... */ }; // B public by default class D7 : protected B { /* ... */ }; struct D8 : protected B { /* ... */ }; //OK with cast
```

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 (7.6.3), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,

```cpp
class B {
  public:
    int mi; // non-static member
    static int si; // static member
};
class D : private B {
};
class DD : public D {
  void f();
};

void DD::f() {
  mi = 3; // error: mi is private in D
  si = 3; // error: si is private in D
  ::B b;
  b.mi = 3; // OK (b.mi is different from this->mi)
  b.si = 3; // OK (b.si is different from this->si)
  ::B::si = 3; // OK
  ::B* bp1 = this; // error: B is a private base class
  ::B* bp2 = (::B*)this;
  bp2->mi = 3; // OK with cast
}
```

116] As specified previously in 11.8, private members of a base class remain inaccessible even to derived classes unless friend declarations within the base class definition are used to grant access explicitly.
A base class \( B \) of \( N \) is accessible at \( R \), if

1. an invented public member of \( B \) would be a public member of \( N \), or
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
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. 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:

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]

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]

1. \( m \) as a member of \( N \) is public, or
2. \( m \) as a member of \( N \) is private, and \( R \) occurs in a member or friend of class \( N \), or
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
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:

class B;
class A {
  private:
    int i;
    friend void f(B*);
};
class B : public A {
};
void f(B* p) {
  p->i = 1;
  // OK: B* can be implicitly converted to A*, and f has access to i in A
}
—end example]

If a class member access operator, including an implicit “this->”, is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the
“.” operator case) cannot be implicitly converted to a pointer to the naming class of the right operand. [Note: This requirement is in addition to the requirement that the member be accessible as named. —end note]

11.8.3 Friends

A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [Example: 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 A {
    class B { }
    friend class X;
};

struct X : A::B {
    // OK: A::B accessible to friend
    A::B mx;
    // OK: A::B accessible to member of friend
    class Y {
        A::B my;
        // OK: A::B accessible to nested member of friend
    };
};
—end example] [Example:

```cpp
class X {
    enum { a=100 }; // OK, Y is a friend of X
};

class Z {
    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]

A friend declaration that does not declare a function shall have one of the following forms:
friend elaborated-type-specifier;
friend simple-type-specifier;
friend typename-specifier;

[Note: A friend declaration may be the declaration in a template-declaration (Clause 13, 13.6.4). — 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 (6.5). Otherwise, the function retains its previous linkage (9.1.1).

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]

A function can be defined in a friend declaration of a class if and only if the class is a non-local class (11.5), 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 function (9.1.6). 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 (6.4.1).

8 No storage-class-specifier shall appear in the decl-specifier-seq of a friend declaration.

9 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 (11.3) portion of the class member-specification.

10 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]

11 If a friend declaration appears in a local class (11.5) 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:

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]

11.8.4 Protected member access [class.protected]

An additional access check beyond those described earlier in 11.8 is applied when a non-static data member
or non-static member function is a protected member of its naming class (11.8.2).117 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:

class B {
  protected:
    int i;
    static int j;
  };

class D1 : public B {
  
};
class D2 : public B {
  friend void fr(B*,D1*,D2*);

117) This additional check does not apply to other members, e.g., static data members or enumerator member constants.

§ 11.8.4 287
void mem(B*, D1*);
);

void fr(B* pb, D1* p1, D2* p2) {
    pb->i = 1; // ill-formed
    p1->i = 2; // ill-formed
    p2->i = 3; // OK (access through a D2)
    p2->B::i = 4; // OK (access through B, even though naming class is B)
    int B::* pmi_B = &B::i; // ill-formed
    int B::* pmi_B2 = &D2::i; // OK (type of &D2::i is int B::*
    B::j = 5; // ill-formed (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; // ill-formed
    p1->i = 2; // ill-formed
    i = 3; // OK (access through this)
    B::i = 4; // OK (access through this, qualification ignored)
    int B::* pmi_B = &B::i; // ill-formed
    int B::* pmi_B2 = &D2::i; // OK
    j = 5; // OK (because j refers to static member)
    B::j = 6; // OK (because B::j refers to static member)
}

void g(B* pb, D1* p1, D2* p2) {
    pb->i = 1; // ill-formed
    p1->i = 2; // ill-formed
    p2->i = 3; // ill-formed
}
—end example]

11.8.5 Access to virtual functions [class.access.virt]

The access rules (11.8) 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
}
—end example]

1 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.

11.8.6 Multiple access [class.paths]

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:
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]

11.8.7 Nested classes [class.access.nest]
1 A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules (11.8) shall be obeyed. [Example:

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]

11.9 Initialization [class.init]
1 When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form (), the object is initialized as specified in 9.3.
2 An object of class type (or array thereof) can be explicitly initialized; see 11.9.1 and 11.9.2.
3 When an array of class objects is initialized (either explicitly or implicitly) and the elements are initialized by constructor, the constructor shall be called for each element of the array, following the subscript order; see 9.2.3.4. [Note: Destructors for the array elements are called in reverse order of their construction. — end note]

11.9.1 Explicit initialization [class.expl.init]
1 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 9.3. [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
```
2 An object of class type can also be initialized by a \textit{braced-init-list}. List-initialization semantics apply; see 9.3 and 9.3.4. [Example:
\begin{verbatim}
complex v[6] = { 1, complex(1,2), complex(), 2 };  
\end{verbatim}
Here, \texttt{complex::complex(double)} is called for the initialization of \texttt{v[0]} and \texttt{v[3]}, \texttt{complex::complex(double, double)} is called for the initialization of \texttt{v[1]}, \texttt{complex::complex()} is called for the initialization \texttt{v[2]}, \texttt{v[4]}, and \texttt{v[5]}. For another example,
\begin{verbatim}
struct X {
  int i;
  float f;
  complex c;
} x = { 99, 88.8, 77.7 };  
\end{verbatim}
Here, \texttt{x.i} is initialized with 99, \texttt{x.f} is initialized with 88.8, and \texttt{complex::complex(double)} is called for the initialization of \texttt{x.c}. — end example]  
[Note: Braces can be elided in the \texttt{initializer-list} for any aggregate, even if the aggregate has members of a class type with user-defined type conversions; see 9.3.1. — 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 \texttt{initializer} is explicitly specified (see 11.9 and 9.3). — end note]
4 [Note: The order in which objects with static or thread storage duration are initialized is described in 6.8.3.3 and 8.7. — end note]

11.9.2 Initializing bases and members  
[\texttt{class.base.init}]

1 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 \textit{ctor-initializer}, which has the form
\begin{verbatim}
ctor-initializer:
  : mem-initializer-list
  mem-initializer-list:
    mem-initializer ...
    mem-initializer-list , mem-initializer ...
  mem-initializer:
    mem-initializer-id ( expression-listopt )
    mem-initializer-id braced-init-list
  mem-initializer-id:
    class-or-decltype
    identifier
\end{verbatim}
2 In a \texttt{mem-initializer-id} an initial unqualified \texttt{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 \texttt{mem-initializer-id} naming the member or base class and composed of a single identifier refers to the class member. A \texttt{mem-initializer-id} for the hidden base class may be specified using a qualified name. — end note] Unless the \texttt{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 \texttt{mem-initializer} is ill-formed.
3 A \texttt{mem-initializer-list} can initialize a base class using any \texttt{class-or-decltype} that denotes that base class type. [Example:
\begin{verbatim}
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{verbatim}
— end example]
4 If a \texttt{mem-initializer-id} is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the \texttt{mem-initializer} is ill-formed. [Example:
\begin{verbatim}
struct A { A(); };  
struct B: public virtual A { }
\end{verbatim}
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:]

```cpp
class C {
    C(int) { } // #1: non-delegating constructor
    C(): C(42) { } // #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
};
```

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 9.3 for direct-initialization. [Example:]

```cpp
class B1 { B1(int); /* ... */ };
class B2 { B2(int); /* ... */ };
class 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) { /* ... */ }
```

A temporary expression bound to a reference member in a *mem-initializer* is ill-formed. [Example:]

```cpp
class A {
    A() : v(42) { } // error
    const int& v;
};
```

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

- if the entity is a non-static data member that has a default member initializer (11.3) and either
  - the constructor’s class is a union (11.4), and no other variant member of that union is designated by a *mem-initializer-id* or
  - 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 9.3;
- otherwise, if the entity is an anonymous union or a variant member (11.4.1), no initialization is performed;
- otherwise, the entity is default-initialized (9.3).
[Note: An abstract class (11.6.3) 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:

```cpp
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]  

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

```cpp
struct A {
    int i = /* some integer expression with side effects */;
    A(int arg) : i(arg) { }
    // ...
};
```

the A(int) constructor will simply initialize i to the value of arg, and the side effects in i’s default member initializer will not take place. — end example]

A temporary expression bound to a reference member from a default member initializer is ill-formed. [Example:

```cpp
struct A {
    A() = default; // OK
    A(int v) : v(v) { } // OK
    const int& v = 42; // OK
};
A a1; // error: ill-formed binding of temporary to reference
A a2(1); // OK, unfortunately
```

— end example]

In a non-delegating constructor, the destructor for each potentially constructed subobject of class type is potentially invoked (11.3.6). [Note: This provision ensures that destructors can be called for fully-constructed subobjects in case an exception is thrown (14.2). — 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 (6.6.2), 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. (regardless of the order of the derived type)

2. Then, direct base classes are initialized in declaration order as they appear in the base-specifier-list (regardless of the order of the derived type).

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 derived type).

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]

14 [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) { /* ... */ }
```

```cpp
V v(1); // use V(int)
A a(2); // use V(int)
B b(3); // use V()
C c(4); // use V()
```

—end example]

15 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]

16 Member functions (including virtual member functions, 11.6.2) 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()) { } // undefined: calls member function but base A not yet initialized
};

class C {
public:
    C(int);
};
class D : public B, C {
    int i;
public:
    D() : C(f()), i(f()) { } // undefined: calls member function but base C not yet initialized
};

—end example

[Note: 11.9.4 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 (13.6.3) 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]

11.9.3 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 (9.8)), 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),

§ 11.9.3 294
```
// 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

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 Z does not invoke default constructor of X

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); // ill-formed: 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() 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.
11.9.4 Construction and destruction  

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:

```c
struct X { int i; }
struct Y : X { Y(); }
struct A { int a; }
struct B : public A { int j; Y y; } // non-trivial

extern B bobj;
B* pb = &bobj;  // OK
int* p1 = &bobj.a;  // undefined, refers to base class member
int* p2 = &bobj.y.i;  // undefined, refers to member's member

A* pa = &bobj;  // undefined, 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,

```c
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 gvalue 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:

```c
struct C;
void no_opt(C*);

struct C {
    int c;
    C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cprr) {
    int i = cobj.c * 100;  // value of cobj.c is unspecified
    cprr->c = 1;
    cout << cobj.c * 100  // value of cobj.c is unspecified
         << '\n';
}

extern struct D d;
struct D {
    D(int a) : a(a), b(d.a) {}  
    int a, b;
};
D d = D(1);  // value of d.b is unspecified
```

—end example]
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: 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
        // but D is not constructed
    X(this) {}              // defined: upon construction of X, C/B/D/A sublattice is fully constructed
};  

—end example]
```

Member functions, including virtual functions (11.6.2), can be called during construction or destruction (11.9.2). 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
    g();              // calls B::g, not D::g
    v->g();          // v is base of B, the call is well-defined, calls B::g
    a->f();          // undefined behavior, a’s type not a base of B
}

—end example]
```

The `typeid` operator (7.6.1.7) can be used during construction or destruction (11.9.2). When `typeid` is used in a constructor (including the `mem-initializer` or default member initializer (11.3) 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.

6 dynamic_cast (7.6.1.6) can be used during construction or destruction (11.9.2). 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: type 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]

11.9.5 Copy/move elision

1 When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class
object, even if the constructor selected for the copy/move operation and/or the destructor for the object
have side effects. In such cases, the implementation treats the source and target of the omitted copy/move
operation as simply two different ways of referring to the same object. If the first parameter of the selected
constructor is an rvalue reference to the object’s type, the destruction of that object occurs when the target
would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects
would have been destroyed without the optimization.118 This elision of copy/move operations, called copy
elision, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

(1.1) — in a return statement in a function with a class return type, when the expression is the name of
a non-volatile automatic object (other than a function parameter or a variable introduced by the
exception-declaration of a handler (14.3)) with the same type (ignoring cv-qualification) as the function
return type, the copy/move operation can be omitted by constructing the automatic 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 automatic object
(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 from the operand to the
exception object (14.1) can be omitted by constructing the automatic object directly into the exception
object

118) Because only one object is destroyed instead of two, and one copy/move constructor is not executed, there is still one object
destroyed for each one constructed.
— in a coroutine (9.4.4), 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

— when the exception-declaration of an exception handler (Clause 14) declares an object of the same type (except for cv-qualification) as the exception object (14.1), 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 (6.8.3.2). [Note: Copy elision might be performed if the same expression is evaluated in another context. — end note]

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 a;
  return a;
}

constexpr A a; // well-formed, a.p points to a
constexpr A b = g(); // well-formed, b.p points to b

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 local automatic object `t` 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 local automatic object to `t2` that is elided. — end example]

In the following copy-initialization contexts, a move operation might be used instead of a copy operation:

— If the expression in a return or co_return statement (8.6.3) is a (possibly parenthesized) id-expression that names an object with automatic storage duration 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 the name of a non-volatile automatic object (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), overload resolution to select the constructor for the copy or the return_value overload to call is first performed as if the object were designated by an rvalue. If the first overload resolution fails or was not performed, or if the type of the first parameter of the selected constructor or the return_value overload is

§ 11.9.5 299
not an rvalue reference to the object’s type (possibly cv-qualified), overload resolution is performed again, considering the object 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 the return_value overload must be accessible even if the call is elided. — end note]

4 [Example:
   ```cpp
   class Thing {
   public:
      Thing();
      ~Thing();
      Thing(Thing&&);
   private:
      Thing(const Thing&);
   }

   Thing t(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]

11.10 Comparisons

11.10.1 Defaulted comparison operator functions

1 A defaulted comparison operator function (7.6.8, 7.6.9, 7.6.10) for some class C shall be a non-template function declared in the member-specification of C that is

(1.1) — a non-static const member of C having one parameter of type const C&, or

(1.2) — a friend of C having two parameters of type const C&.

2 If the class definition does not explicitly declare an == operator function, but declares a defaulted three-way comparison operator function, an == operator function is declared implicitly with the same access as the three-way comparison operator function. The implicitly-declared == operator for a class X is an inline member and is defined as defaulted in the definition of X. If the three-way comparison operator function is declared as a non-static const member, the implicitly-declared == operator function is a member of the form

   ```cpp
   bool X::operator==(const X&) const;
   ```

Otherwise, the implicitly-declared == operator function is of the form

   ```cpp
   friend bool operator==(const X&, const X&);
   ```

   [Note: Such a friend function is visible to argument-dependent lookup (6.4.2) only (9.7.1.2). — end note]

   The operator is a constexpr function if its definition would satisfy the requirements for a constexpr function.

   [Note: The == operator function is declared implicitly even if the defaulted three-way comparison operator function is defined as deleted. — end note]

3 A type C has strong structural equality if, given a glvalue x of type const C, either:

(3.1) — C is a non-class type and x <=> x is a valid expression of type std::strong_ordering or std::strong_
C is a class type with an == operator defined as defaulted in the definition of C. x == x is well-formed when contextually converted to bool, all of C’s base class subobjects and non-static data members have strong structural equality, and C has no mutable or volatile subobjects.

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 (12.3.3.1), class member access expressions (7.6.1.4), and array subscript expressions (7.6.1.1) applied to x. It is unspecified whether virtual base class subobjects appear more than once in the expanded list of subobjects.

11.10.2 Equality operators

A defaulted equality operator (7.6.10) function shall have a declared return type bool.

A defaulted == operator function for a class C is defined as deleted unless, for each x_i in the expanded list of subobjects for an object x of type C, x_i == x_i is a valid expression and contextually convertible to bool.

The return value V of a defaulted == operator function with parameters x and y is determined by comparing corresponding elements x_i and y_i in the expanded lists of subobjects for x and y (in increasing index order) until the first index i where x_i == y_i yields a result value which, when contextually converted to bool, yields false. If no such index exists, V is true. Otherwise, V is false.

A defaulted != operator function for a class C with parameters x and y is defined as deleted if

- overload resolution (12.3), as applied to x == y (also considering synthesized candidates with reversed order of parameters (12.3.1.2)), results in an ambiguity or a function that is deleted or inaccessible from the operator function, or

- x == y cannot be contextually converted to bool.

Otherwise, the operator function yields (x == y) ? false : true.

[Example:

```cpp
struct D {
    int i;
    friend bool operator==(const D& x, const D& y) = default;
    // OK, returns x.i == y.i
    bool operator!=(const D& z) const = default; // OK, returns (*this == z) ? false : true
};

— end example]

11.10.3 Three-way comparison

Given an expanded list of subobjects for an object x of type C, the type of the expression x_i <=> x_i is denoted by R_i. If the declared return type of a defaulted three-way comparison operator function is auto, then the return type is deduced as the common comparison type (see below) of R_0, R_1, ..., R_{n−1}. [Note: Otherwise, the program will be ill-formed if the expression x_i <=> x_i is not implicitly convertible to the declared return type for any i. — end note] If the return type is deduced as void, the operator function is defined as deleted.

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 x_i <=> y_i yields a result value v_i where v_i != 0, contextually converted to bool, yields true; V is v_i, converted to R. If no such index exists, V is std::strong_ordering::equal converted to R.

The common comparison type U of a possibly-empty list of n types T_0, T_1, ..., T_{n−1} is defined as follows:

1. If any T_i is not a comparison category type (17.11.2), U is void.
2. Otherwise, if at least one T_i is std::weak_equality, or at least one T_i is std::strong_equality and at least one T_i is std::partial_ordering or std::weak_ordering, U is std::weak_equality (17.11.2.2).
3. Otherwise, if at least one T_i is std::strong_equality, U is std::strong_equality (17.11.2.3).
4. Otherwise, if at least one T_i is std::partial_ordering, U is std::partial_ordering (17.11.2.4).
5. Otherwise, if at least one T_i is std::weak_ordering, U is std::weak_ordering (17.11.2.5).
Otherwise, \( U \) is `std::strong_ordering` (17.11.2.6). [Note: In particular, this is the result when \( n \) is 0. — end note]

### 11.10.4 Relational operators

1. A defaulted relational (7.6.9) operator function for some operator \( @ \) shall have a declared return type `bool`.

2. The operator function with parameters \( x \) and \( y \) is defined as deleted if

   - overload resolution (12.3), as applied to \( x <=> y \) results in an ambiguity or a function that is deleted or inaccessible from the operator function, or

   - the operator \( @ \) cannot be applied to the return type of \( x <=> y \).

3. Otherwise, the operator function yields \( x <=> y @ 0 \).

4. [Example:

   ```c++
   struct C {
     friend std::strong_equality operator<=>(const C&, const C&);
     bool operator<(const C&) const = default; // OK, function is deleted
   };
   — end example]

### 11.11 Free store

1. Any allocation function for a class \( T \) is a static member (even if not explicitly declared `static`).

2. [Example:

   ```c++
   class Arena;
   struct B {
     void* operator new(std::size_t, Arena*);
   };
   struct D1 : B {
   };
   Arena* ap;
   void foo(int i) {
     new (ap) D1; // calls B::operator new(std::size_t, Arena*)
     new D1[i]; // calls ::operator new[](std::size_t)
     new D1; // ill-formed: ::operator new(std::size_t) hidden
   } — end example]

3. When an object is deleted with a `delete-expression` (7.6.2.6), a deallocation function (`operator delete()`) for non-array objects or `operator delete[]()` for arrays is (implicitly) called to reclaim the storage occupied by the object (6.6.5.4.2).

4. Class-specific deallocation function lookup is a part of general deallocation function lookup (7.6.2.6) and occurs as follows. If the `delete-expression` is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is the one selected at the point of definition of the dynamic type’s virtual destructor (11.3.6). 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 (7.6.2.6) continues. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

5. Any deallocation function for a class \( X \) is a static member (even if not explicitly declared `static`). [Example:

   ```c++
   class X {
     void operator delete(void*);
     void operator delete[](void*, std::size_t);
   };
   class Y {
     void operator delete(void*, std::size_t);
   }
   ```

119) A similar provision is not needed for the array version of `operator delete` because 7.6.2.6 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.
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*, std::destroying_delete_t) {
        p->log_deletion();
        p->~E();
        ::operator delete(p);
    }
};

void f() {
    B* bp = new D;
    delete bp;    // uses D::operator delete(void*)
    bp = new E;
    delete bp;    // uses E::operator delete(E*, std::destroying_delete_t)
}
```

Here, storage for the object of class D is deallocated by D::operator delete(), and the object of class E is destroyed and its storage is deallocated by E::operator delete(), due to the virtual destructor. — end note] [Note: 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);
};

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]

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]

[Note: If a deallocation function has no explicit noexcept-specifier, it has a non-throwing exception specification (14.4). — end note]
12 Overloading

1 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.

2 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 12.3. [Example:

```c
double abs(double);
int abs(int);

abs(1);         // calls abs(int);
abs(1.0);       // calls abs(double);
```

—end example]

12.1 Overloadable declarations

1 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 (9.8). 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]

2 Certain function declarations cannot be overloaded:

(2.1) — Function declarations that differ only in the return type, the exception specification (14.4), or both cannot be overloaded.

(2.2) — Member function declarations with the same name and the same parameter-type-list (9.2.3.5) cannot be overloaded if any of them is a static member function declaration (11.3.8). Likewise, member function template declarations with the same name, the same parameter-type-list, and the same template parameter lists 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 (12.3.1) 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 and the same parameter-type-list, 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();     // ok
  void f();            // ok
  void f() const;      // ok
  void f() const volatile; // ok
  void g();            // ok
  void g() const;      // ok: no static g
  void g() const volatile; // ok: no static g
};
```

—end example]

(2.3) — Member function declarations with the same name and the same parameter-type-list (9.2.3.5) as well as member function template declarations with the same name, the same parameter-type-list, and the same template parameter lists cannot be overloaded if any of them, but not all, have a ref-qualifier (9.2.3.5). [Example:

```c
class Y {
  void h() &;        // ok
  void h() const &;  // ok
  void h() &&;       // ok, all declarations have a ref-qualifier
};
```
void i() &;
void i() const; // ill-formed, prior declaration of i
                // has a ref-qualifier
};

— end example

3 [Note: As specified in 9.2.3.5, function declarations that have equivalent parameter declarations and requires-clauses, if any (13.4.2), 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 (9.1.3). [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)
— end example]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [Example:

def E { a ;
void f(E i) { /* ... */ }
void f(int 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 (9.2.3.5). Only the second and subsequent array dimensions are significant in parameter types (9.2.3.4). [Example:

int f(char[]);
int f(char[*][10]); // same as f(char[*]);
int f(char[*][]); // 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 (9.2.3.5). [Example:

void h(int());
void h(int (*)(())); // redeclaration of h(int())
void h(int x()); {}
    // definition of h(int())
void h(int (*)(x)()); {}
    // ill-formed: 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 (const int);
int f (int); // redeclaration of f(int)
int f (int) { /* ... */ } // definition of f(int)
int f (cInt) { /* ... */ } // error: redefinition of f(int)
— end example]
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”.

(3.5) 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 ();                   // OK: call f(int, int)
    f ();                   // error: f(int, int) or f()?
}

— end example]
— end note]

12.2 Declaration matching [over.dcl]

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (12.1) and equivalent trailing requires-clauses, if any (9.2). 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();  // OK
    pd->f("Ben");  // OK, calls D::f
}

— end example]

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);
    f("asdf");  // error: f(int) hides f(const char*)
    // so there is no f(const char*) in this scope
}

void caller () {
    extern void callee(int, int);
    {
        extern void callee(int);  // hides callee(int, int)
        callee(88, 99);  // error: only callee(int) in scope
    }
```

120) When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.
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]

12.3 Overload resolution [over.match]

1

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]

2

Overload resolution selects the function to call in seven distinct contexts within the language:

(2.1) — invocation of a function named in the function call syntax (12.3.1.1.1);

(2.2) — invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax (12.3.1.1.2);

(2.3) — invocation of the operator referenced in an expression (12.3.1.2);

(2.4) — invocation of a constructor for default- or direct-initialization (9.3) of a class object (12.3.1.3);

(2.5) — invocation of a user-defined conversion for copy-initialization (9.3) of a class object (12.3.1.4);

(2.6) — invocation of a conversion function for initialization of an object of a non-class type from an expression of class type (12.3.1.5); and

(2.7) — invocation of a conversion function for conversion in which a reference (9.3.3) will be directly bound (12.3.1.6).

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:

(2.8) — First, a subset of the candidate functions (those that have the proper number of arguments and meet certain other conditions) is selected to form a set of viable functions (12.3.2).

(2.9) — Then the best viable function is selected based on the implicit conversion sequences (12.3.3.1) needed to match each argument to the corresponding parameter of each viable function.

3

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 (11.8) in the context in which it is used, the program is ill-formed.

12.3.1 Candidate functions and argument lists [over.match.funcs]

1

The subclauses of 12.3.1 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.
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.

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.

For non-static member functions, the type of the implicit object parameter is

(4.1) — “lvalue reference to cv X” for functions declared without a ref-qualifier or with the & ref-qualifier

(4.2) — “rvalue reference to cv X” for functions declared with the && ref-qualifier

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 const member function of class X, the extra parameter is assumed to have type “reference to 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 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 (12.3.3). — end note]

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 ref-qualifier, an additional rule applies:

(5.1) — even if the implicit object parameter is not const-qualified, an 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 rvalue does not affect the ranking of implicit conversion sequences (12.3.3.2). — end note]

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 (12.3.3, 12.3.3.1). [Example:

```c++
class T {
public:
  T();
};

class C : T {
public:
  C(int);
};
T a = 1; // ill-formed: T(C(1)) not tried
```

— end example]

In each case where a candidate is a function template, candidate function template specializations are generated using template argument deduction (13.9.3, 13.9.2). If a constructor template or conversion function template has an explicit-specifier whose constant-expression is value-dependent (13.7.2), template argument deduction is performed first and then, if the context requires a candidate that is not explicit and the generated specialization is explicit (9.1.2), 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 overloaded non-template functions. In such a case, the candidate functions generated from each function template are combined with the set of non-template candidate functions.

A defaulted move special function (11.3.4.2, 11.3.5) that is defined as deleted is excluded from the set of candidate functions in all contexts. A constructor inherited from class type C (11.9.3) that has a first

[121] The process of argument deduction fully determines the parameter types of the function template specializations, i.e., the parameters of function template specializations contain no template parameter types. Therefore, except where specified otherwise, function template specializations and non-template functions (9.2.3.5) are treated equivalently for the remainder of overload resolution.
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:

```cpp
struct A {
    A();
    A(A &&); // #1
    template<typename T> A(T &&); // #2
};
struct B : A {
    using A::A;
    B(const B &); // #3
    B(B &&) = default; // #4, implicitly deleted
    struct X { X(X &&) = delete; } x;
};
extern B b1;
B b2 = static_cast<B&&>(b1); // calls #3: #1, #2, and #4 are not viable
struct C { operator B&&(); }; B b3 = C(); // calls #3
—end example]

12.3.1.1 Function call syntax [over.match.call]

1 In a function call (7.6.1.2)
```
prefix-expression ( expression-list_opt )
```
if the \textit{prefix-expression} denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 12.3.1.1.1. If the \textit{prefix-expression} denotes an object of class type, overload resolution is applied as specified in 12.3.1.1.2.

2 If the \textit{prefix-expression} denotes the address of a set of overloaded functions and/or function templates, 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 12.4. —end note]

12.3.1.1.1 Call to named function [over.call.func]

1 Of interest in 12.3.1.1.1 are only those function calls in which the \textit{prefix-expression} ultimately contains a name that denotes one or more functions that might be called. Such a \textit{prefix-expression}, perhaps nested arbitrarily deep in parentheses, has one of the following forms:
```
prefix-expression:
prefix-expression . id-expression
prefix-expression -> id-expression
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{-} 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{prefix-expression} that is the left operand of the \texttt{.} operator has type \texttt{“cv T”} where \texttt{T} denotes a class.\textsuperscript{122} Under this assumption, the \textit{id-expression} in the call is looked up as a member function of \texttt{T} following the rules for looking up names in classes (11.7). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the \textit{expression-list} in the call augmented by the addition of the left operand of the \texttt{.} operator in the normalized member function call as the implied object argument (12.3.1).

3 In unqualified function calls, the name is not qualified by an \texttt{-} or \texttt{.} operator and has the more general form of a \textit{primary-expression}. The name is looked up in the context of the function call following the normal rules for name lookup in expressions (6.4). 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

\textsuperscript{122} Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.
12.3.1.2 Call to object of class type

If the `postfix-expression` `E` in the function call syntax evaluates to a class object of type “`cv T`”, then the set of candidate functions includes at least the function call operators of `T`. The function call operators of `T` are obtained by ordinary lookup of the name `operator()` in the context of `(E).operator()`.

In addition, for each non-explicit conversion function declared in `T` of the form

    operator conversion-type-id () cv-qualifier ref-qualifier opt noexcept-specifier opt attribute-specifier-seqopt;

where `cv-qualifier` 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`,...,`P_n`) returning `R`”, or the type “reference to pointer to function of (`P_1`,...,`P_n`) returning `R`”, or the type “reference to function of (`P_1`,...,`P_n`) returning `R`”, a surrogate call function with the unique name `call-function` and having the form

    `R` call-function (`conversion-type-id` `F`, `P_1` `a_1`, ..., `P_n` `a_n`) { return `F`(`a_1`, ..., `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.

If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert `E` to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call. If the conversion function cannot be called (e.g., because of an ambiguity), the program is ill-formed.

The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument (`E`). [Note: 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:

```c
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
```

12.3.1.2 Operators in expressions

If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to 7.6. [Note: 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* () const;
};

String operator + (const String&, const String&);

void f() {
    const char* p = "one" + "two";  // ill-formed because neither operand has class or enumeration type
    int I = 1 + 1;                 // always evaluates to 2 even if class or enumeration types exist
    // that would perform the operation.
}

/* end example */

If either operand has a type that is a class or an enumeration, a user-defined operator function 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 13 (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 13 — 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>12.5.1</td>
<td>@a</td>
<td>(a).operator@()</td>
<td>operator@a()</td>
</tr>
<tr>
<td>12.5.2</td>
<td>a0b</td>
<td>(a).operator@(b)</td>
<td>operator,a,b</td>
</tr>
<tr>
<td>12.5.3</td>
<td>a+b</td>
<td>(a).operator=()</td>
<td>operator=a,b</td>
</tr>
<tr>
<td>12.5.5</td>
<td>a[b]</td>
<td>(a).operator<a href="b"></a></td>
<td>operator(a,b)</td>
</tr>
<tr>
<td>12.5.6</td>
<td>a-&gt;</td>
<td>(a).operator-&gt;()</td>
<td></td>
</tr>
<tr>
<td>12.5.7</td>
<td>a@</td>
<td>(a).operator@()</td>
<td>operator(a,0)</td>
</tr>
</tbody>
</table>

For a unary operator @ with an operand of a type whose cv-unqualified version is T1, and for a binary operator @ with a left operand of a type whose cv-unqualified version is T1 and a right operand of a type whose cv-unqualified version is 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@ (12.3.1.1.1); 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 (6.4.2) 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 12.6 that, compared to the given operator,

(3.1.1) have the same operator name, and

(3.1.2) accept the same number of operands, and

(3.1.3) accept operand types to which the given operand or operands can be converted according to 12.3.3.1, and

(3.1.4) do not have the same parameter-type-list as any non-member candidate that is not a function template specialization.

(3.4) For the relational (7.6.9) operators, the rewritten candidates include all member, non-member, and built-in candidates for the operator <=> for which the rewritten expression (x <=> y) @ 0 is well-formed.
using that operator<=>. 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 member, non-member, and built-in candidate for the operator <=> for which the rewritten expression 0 @ (y <=> x) is well-formed using that operator<=>. For the != operator (7.6.10), the rewritten candidates include all member, non-member, and built-in candidates for the operator == for which the rewritten expression (x == y) is well-formed when contextually converted to bool using that operator==. For the equality operators, the rewritten candidates also include a synthesized candidate, with the order of the two parameters reversed, for each member, non-member, and built-in candidate for the operator == for which the rewritten expression (y == x) is well-formed when contextually converted to bool using that operator==. [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] In each case, rewritten candidates are not considered in the context of the rewritten expression. For all other operators, the rewritten candidate set is empty.

For the built-in assignment operators, conversions of the left operand are restricted as follows:

(4.1) — no temporaries are introduced to hold the left operand, and

(4.2) — no user-defined conversions are applied to the left operand to achieve a type match with the left-most parameter of a built-in candidate.

For all other operators, no such restrictions apply.

The set of candidate functions for overload resolution for some operator @ is the union of the member candidates, the non-member candidates, the built-in candidates, and the rewritten candidates for that operator @.

The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 12.3.2 and 12.3.3. [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]

If a rewritten candidate is selected by overload resolution for a relational or three-way comparison operator @, x @ y is interpreted as the rewritten expression: 0 @ (y <=> x) if the selected candidate is a synthesized candidate with reversed order of parameters, or (x <=> y) @ 0 otherwise, using the selected rewritten operator<=> candidate. If a rewritten candidate is selected by overload resolution for a != operator, x != y is interpreted as (y == x) ? false : true if the selected candidate is a synthesized candidate with reversed order of parameters, or (x == y) ? false : true otherwise, using the selected rewritten operator== candidate. If a rewritten candidate is selected by overload resolution for an == operator, x == y is interpreted as (y == x) ? true : false using the selected rewritten operator== candidate.

If a built-in candidate is selected by overload resolution, the operands of class type are converted to the types of the corresponding parameters of the selected operation function, except that the second standard conversion sequence of a user-defined conversion sequence (12.3.3.1.2) is not applied. Then the operator is treated as the corresponding built-in operator and interpreted according to 7.6. [Example:

```c
struct X {
    operator double();
};
struct Y {
    operator int*();
};
```

125) If the set of candidate functions is empty, overload resolution is unsuccessful.

§ 12.3.1.2
The second operand of operator \( \to \) is ignored in selecting an \texttt{operator\to} function, and is not an argument when the \texttt{operator\to} function is called. When \texttt{operator\to} returns, the operator \( \to \) is applied to the value returned, with the original second operand.\(^{126}\)

If the operator is the operator \( \_\_\_\_ \), the unary operator \&, or the operator \( \to \); and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to 7.6.\(^{12} \)

[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:

```c
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]

12.3.1.3 Initialization by constructor \([\texttt{over.match.ctor}]\)

When objects of class type are direct-initialized (9.3), copy-initialized from an expression of the same or a derived class type (9.3), or default-initialized (9.3), 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 (11.3.7.1) of that class. The argument list is the \texttt{expression-list} or \texttt{assignment-expression} of the \texttt{initializer}.

12.3.1.4 Copy-initialization of class by user-defined conversion \([\texttt{over.match.copy}]\)

Under the conditions specified in 9.3, 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 “\texttt{cv1 T}” is the type of the object being initialized, with \texttt{T} a class type, the candidate functions are selected as follows:

\begin{enumerate}
\item When the type of the initializer expression is a class type “\texttt{cv S}”, the non-explicit conversion functions of \texttt{S} and its base classes are considered. When initializing a temporary object (11.3) to be bound to the first parameter of a constructor where the parameter is of type “reference to possibly \texttt{cv}-qualified \texttt{T}” and the constructor is called with a single argument in the context of direct-initialization of an object of type “\texttt{cv2 T}”, explicit conversion functions are also considered. Those that are not hidden within \texttt{S} and yield a type whose cv-unqualified version is the same type as \texttt{T} or is a derived class thereof are candidate functions. A call to a conversion function returning “reference to \texttt{X}” is a glvalue of type \texttt{X}, and such a conversion function is therefore considered to yield \texttt{X} for this process of selecting candidate functions.
\end{enumerate}

\(^{126}\) If the value returned by the \texttt{operator\to} function has class type, this may result in selecting and calling another \texttt{operator\to} function. The process repeats until an \texttt{operator\to} function returns a value of non-class type.
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]

12.3.1.5 Initialization by conversion function

Under the conditions specified in 9.3, 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:

(1.1) — 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 (12.3.3.1.1) 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.

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]

12.3.1.6 Initialization by conversion function for direct reference binding

Under the conditions specified in 9.3.3, 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:

(1.1) — 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 “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 lvalue reference or an lvalue reference to function), where “cv1 T” is reference-compatible (9.3.3) with “cv2 T2”, are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within S and yield type “cv2 T2” (when initializing an lvalue reference or an lvalue reference to function) or “rvalue reference to cv2 T2” (when initializing an lvalue 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.

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]

12.3.1.7 Initialization by list-initialization

When objects of non-aggregate class type T are list-initialized such that 9.3.4 specifies that overload resolution is performed according to the rules in this subclause, overload resolution selects the constructor in two phases:

(1.1) — Initially, the candidate functions are the initializer-list constructors (9.3.4) of the class T and the argument list consists of the initializer list as a single argument.

(1.2) — If no viable initializer-list constructor is found, overload resolution is performed again, where the candidate functions are all the constructors of the class T and the argument list consists of the elements of the initializer list.

If the initializer list has no elements and T has a default constructor, the first phase is omitted. In copy-list-initialization, if an explicit constructor is chosen, the initialization is ill-formed. [Note: This differs from other situations (12.3.1.3, 12.3.1.4), where only converting constructors are considered for copy-initialization. This restriction only applies if this initialization is part of the final result of overload resolution. —end note]

12.3.1.8 Class template argument deduction

When resolving a placeholder for a deduced class type (9.1.7.6) where the template-name names a primary class template C, a set of functions and function templates is formed comprising:

(1.1) — 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.

If C is not defined or does not declare any constructors, an additional function template derived as above from a hypothetical constructor C().

An additional function template derived as above from a hypothetical constructor C(C), called the copy deduction candidate.

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.

Initialization and overload resolution are performed as described in 9.3 and 12.3.1.3, 12.3.1.4, or 12.3.1.7 (as appropriate for the type of initialization performed) for an object of a hypothetical class type, where the selected functions and function templates 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. As an exception, the first phase in 12.3.1.7 (considering initializer-list constructors) is omitted if the initializer list consists of a single expression of type cv U, where U is a specialization of C or a class derived from a specialization of C. 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 realue 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*> — end example
```

12.3.2 Viable functions

From the set of candidate functions constructed for a given context (12.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences and associated constraints (13.4.2) for the best fit (12.3.3). 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 (9.2.3.5). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to “match the ellipsis” (12.3.3.1.3).

(2.3) A candidate function having more than \(m\) parameters is viable only if the \((m+1)\)th parameter has a default argument (9.2.3.6). 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 (13.4.2), those constraints shall be satisfied (13.4.1).

Third, for \(F\) to be a viable function, there shall exist for each argument an implicit conversion sequence (12.3.3.1) 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 12.3.3.1.4).

12.3.3 Best viable function \([\text{over.match.best}]\)

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 \(\text{ICS}_1(F)\); otherwise,

(1.2) let \(\text{ICS}_i(F)\) denote the implicit conversion sequence that converts the \(i\)-th argument in the list to the type of the \(i\)-th parameter of viable function \(F\). 12.3.3.1 defines the implicit conversion sequences and 12.3.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

Given these definitions, a viable function \(F_1\) is defined to be a better function than another viable function \(F_2\) if for all arguments \(i\), \(\text{ICS}_i(F_1)\) 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 9.3, 12.3.1.5, and 12.3.1.6) 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 \([\text{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 (12.3.1.6) 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 \([\text{Example:}]

```
template <class T> struct A {
  operator T&(); // #1
```
operator T&&(); // #2
};
typedef int Fn();
A<Fn> a;
Fn& lf = a; // calls #1
Fn&& rf = a; // calls #2

— 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 13.6.6.2, 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 13.4.4, 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:

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 (12.3.1.2) and F1 is not [Example:

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:

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 (12.3.1.8) and F2 is not, or, if not that,

— (2.11) F1 is the copy deduction candidate (12.3.1.8) 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:

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
void Fcn(const int*, short);
void Fcn(int*, int);

int i;
short s = 0;

void f() {
  Fcn(&i, s);  // is ambiguous because &i → int* is better than &i → const int*
  // but s → short is also better than s → int
  Fcn(&i, 1L);  // calls Fcn(int*, int), because &i → int* is better than &i → const int*
  // and 1L → short and 1L → int are indistinguishable
  Fcn(&i, 'c');  // calls Fcn(int*, int), because &i → int* is better than &i → const int*
  // and c → int is better than c → short
}
— end example]
```

4 If the best viable function resolves to a function for which multiple declarations were found, and if at least two of these declarations — or the declarations they refer to in the case of using-declarations — specify a default argument that made the function viable, the program is ill-formed. [Example:

```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]
```

### 12.3.3.1 Implicit conversion sequences

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

[129] 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.
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 (9.3, 9.3.3).

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 (9.4.3), 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 (12.3.3.1.1),
2. a user-defined conversion sequence (12.3.3.1.2), or
3. an ellipsis conversion sequence (12.3.3.1.3).

However, if the target is

- the first parameter of a constructor or
- the implicit object parameter of a user-defined conversion function

and the constructor or user-defined conversion function is a candidate by

- 12.3.1.3, when the argument is the temporary in the second step of a class copy-initialization,
- 12.3.1.4, 12.3.1.5, or 12.3.1.6 (in all cases), or
- the second phase of 12.3.1.7 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:

```c
struct Y { Y(int); };  
struct A { operator int() const{}; }; 
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

—end example]

For the case where the parameter type is a reference, see 12.3.3.1.4.

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 (12.3.3.1.1).

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 (12.3.3.1.1).
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 12.3.3.2, 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); // ill-formed: ambiguous because there is a conversion b \rightarrow C (via constructor)  
// and an (ambiguous) conversion b \rightarrow 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.

12.3.3.1.1 Standard conversion sequences

Table 14 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]

Each conversion in Table 14 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (12.3.3.2). The rank of a conversion sequence is determined by considering the rank of each conversion in the sequence and the rank of any reference binding (12.3.3.1.4). 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.

12.3.3.1.2 User-defined conversion sequences

A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (11.3.7) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (11.3.7.1), 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 (11.3.7.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

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 12.3.3 and 12.3.3.1).

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.
<table>
<thead>
<tr>
<th>Conversion</th>
<th>Category</th>
<th>Rank</th>
<th>Subclause</th>
</tr>
</thead>
<tbody>
<tr>
<td>No conversions required</td>
<td>Identity</td>
<td></td>
<td>7.3.1</td>
</tr>
<tr>
<td>Lvalue-to-rvalue conversion</td>
<td>Lvalue Transformation</td>
<td>Exact Match</td>
<td></td>
</tr>
<tr>
<td>Array-to-pointer conversion</td>
<td></td>
<td>7.3.2</td>
<td></td>
</tr>
<tr>
<td>Function-to-pointer conversion</td>
<td></td>
<td>7.3.3</td>
<td></td>
</tr>
<tr>
<td>Qualification conversions</td>
<td></td>
<td>7.3.5</td>
<td></td>
</tr>
<tr>
<td>Function pointer conversion</td>
<td></td>
<td>7.3.13</td>
<td></td>
</tr>
<tr>
<td>Integral promotions</td>
<td>Promotion</td>
<td>Promotion</td>
<td>7.3.6</td>
</tr>
<tr>
<td>Floating-point promotion</td>
<td></td>
<td>7.3.7</td>
<td></td>
</tr>
<tr>
<td>Integral conversions</td>
<td></td>
<td>7.3.8</td>
<td></td>
</tr>
<tr>
<td>Floating-point conversions</td>
<td></td>
<td>7.3.9</td>
<td></td>
</tr>
<tr>
<td>Floating-integral conversions</td>
<td></td>
<td>7.3.10</td>
<td></td>
</tr>
<tr>
<td>Pointer conversions</td>
<td></td>
<td>7.3.11</td>
<td></td>
</tr>
<tr>
<td>Pointer-to-member conversions</td>
<td></td>
<td>7.3.12</td>
<td></td>
</tr>
<tr>
<td>Boolean conversions</td>
<td></td>
<td>7.3.14</td>
<td></td>
</tr>
</tbody>
</table>

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.

12.3.3.1.3 Ellipsis conversion sequences

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).

12.3.3.1.4 Reference binding

When a parameter of reference type binds directly (9.3.3) 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 (12.3.3.1).

Example:
```c
struct A {};
struct B : public A {} b;
int f(&);
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 (12.3.3.1.2), 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.

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 12.3.3.1. 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.

Except for an implicit object parameter, for which see 12.3.1, 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 9.3.3). — end note]

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 “value 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,
12.3.3.1.5  List-initialization sequence

When an argument is an initializer list (9.3.4), it is not an expression and special rules apply for converting it to a parameter type.

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 (9.3.1), 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 (9.3.4).] [Example:

```c
struct A { int x, y; }
struct B { int y, x; }

void f(A a, int); // #1
void f(B b, ...); // #2
void g(A a); // #3
void g(B b); // #4
void h() {
  f({.x = 1, .y = 2}, 0); // OK; calls #1
  f({.y = 2, .x = 1}, 0); // error: selects #1, initialization of a fails
                    // due to non-matching member order (9.3.4)
  g({.x = 1, .y = 2}); // error: ambiguous between #3 and #4
}
```

—end example] —end note]

Otherwise, if the parameter type is an aggregate class \( X \) and the initializer list has a single element of type \( \text{cv} \ U \), 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.

Otherwise, if the parameter type is a character array and the initializer list has a single element that is an appropriately-typed string literal (9.3.2), the implicit conversion sequence is the identity conversion.

Otherwise, if the parameter type is \( \text{std}::\text{initializer}\_\text{list}<X> \) 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:

```c
void f(std::initializer_list<int>);
f( {} ); // OK: f(initializer_list<int>) identity conversion
f( {1,2,3} ); // OK: f(initializer_list<int>) identity conversion
f( {'a','b'} ); // OK: f(initializer_list<int>) integral promotion
f( {1.0} ); // error: narrowing

struct A {
  A(std::initializer_list<double>); // #1
  A(std::initializer_list<complex<double>>); // #2
  A(std::initializer_list<std::string>); // #3
};
A a{ 1.0,2.0 }; // OK, uses #1

void g(A);
g({ "foo", "bar" }); // OK, uses #3

typedef int IA[3];
void h(const IA&);
h({ 1, 2, 3 }); // OK: identity conversion
```

—end example]

130) Since there are no parameters of array type, this will only occur as the referenced type of a reference parameter.

§ 12.3.3.1.5
Otherwise, if the parameter type is “array of N X”, if there exists an implicit conversion sequence for each element of the array from the corresponding element of the initializer list (or from {} if there is no such element), the implicit conversion sequence is the worst such implicit conversion sequence.

Otherwise, if the parameter is a non-aggregate class X and overload resolution per 12.3.1.7 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 12.3.3.1. [Example:

```c++
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)
```

—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 (9.3.1), the implicit conversion sequence is a user-defined conversion sequence with the second standard conversion sequence an identity conversion. [Example:

```c++
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]

Otherwise, if the parameter is a reference, see 12.3.3.1.4. [Note: The rules in this subclause will apply for initializing the underlying temporary for the reference. — end note] [Example:

```c++
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
10  
Otherwise, if the parameter type is not a class:

(10.1) — if the initializer list has one element that is not itself an initializer list, the implicit conversion sequence is the one required to convert the element to the parameter type; [Example:

    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]

11 In all cases other than those enumerated above, no conversion is possible.

12.3.3.2 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 12.3.3.1)

(2.1) — a standard conversion sequence (12.3.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

(2.2) — a user-defined conversion sequence (12.3.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (12.3.3.1.3).

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

(3.1) — List-initialization sequence L1 is a better conversion sequence than list-initialization sequence L2 if

(3.1.1) — L1 converts to std::initializer_list<X> for some X and L2 does not, or, if not that,

(3.1.2) — L1 converts to type “array of N1 T”, L2 converts to type “array of N2 T”, and N1 is smaller than N2, even if one of the other rules in this paragraph would otherwise apply. [Example:

    void f1(int);
    void f1(std::initializer_list<long>);  // #1
    void g1() { f1(42); }  // #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]

(3.2) — Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

§ 12.3.3.2  324
(3.2.1) — S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form defined by 12.3.3.1.1, 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 (9.3.3) 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:

```cpp
int i;
int f1();
int&& f2();
int g(const int&);
int g(const int&&);
ing(i);
int k = g(f1());
ing(j);
int l = g(f2());
```

```cpp
struct A {
    && operator<<(int);
    void p() &;
    void p() &&;
};
&& operator<<(&&, char);
A() << 1;  // calls A::operator<<(int)
A() << 'c'; // calls A::operator<<(&&, char)
a;
a << 1;    // calls A::operator<<(int)
a << 'c'; // calls A::operator<<(int)
A().p();  // calls A::p()&
a.p();    // calls A::p()&
```

— end example] or, if not that,

(3.2.4) — S1 and S2 include reference bindings (9.3.3) and S1 binds an lvalue reference to a function lvalue and S2 binds an rvalue reference to a function lvalue [Example:

```cpp
int f(void(&)()); // #1
int f(void(&&)()); // #2
void g();
int i1 = f(g);    // calls #1
```

— end example] or, if not that,

(3.2.5) — S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (7.3.5), respectively, and the cv-qualification signature of type T1 is a proper subset of the cv-qualification signature of type T2 [Example:

```cpp
int f(const volatile int *);
int f(const int *);
int i;
int j = f(&i); // calls f(const int*)
```

— end example] or, if not that,

(3.2.6) — S1 and S2 include reference bindings (9.3.3), 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:

```cpp
int f(const int &);
int f(int &);
g(const int &);
g(int);
```

```cpp
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 12.3.3); in all other contexts, the source types will be the same and the target types will be different. —end note]
12.4 Address of overloaded function

A use of an overloaded function name without arguments is resolved in certain contexts to a function, a pointer to function or a pointer to member function for a specific function from the overload set. A function template name is considered to name a set of overloaded functions in such contexts. A function with type \( F \) is selected for the function type \( FT \) of the target type required in the context if \( (F \text{ after possibly applying the function pointer conversion (7.3.13)}) \) 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] The target can be

1. an object or reference being initialized (9.3, 9.3.3, 9.3.4),
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 (12.5),
5. the return value of a function, operator function, or conversion (8.6.3),
6. an explicit type conversion (7.6.1.3, 7.6.1.8, 7.6.3), or
7. a non-type template-parameter (13.3.2).

The overloaded function name can be preceded by the & operator. An overloaded function name shall not be used without arguments in contexts other than those listed. [Note: Any redundant set of parentheses surrounding the overloaded function name is ignored (7.5.3). — end note]

If the name is a function template, template argument deduction is done (13.9.2.2), 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 overloaded functions considered. [Note: As described in 13.9.1, if deduction fails and the function template name is followed by an explicit template argument list, the template-id is then examined to see whether it identifies a single function template specialization. If it does, the template-id is considered to be an lvalue for that function template specialization. The target type is not used in that determination. — end note]

Non-member functions and static member functions match targets of function pointer type or reference to function type. Non-static member functions match targets of pointer-to-member-function type. If a non-static member function is selected, the reference to the overloaded function name is required to have the form of a pointer to member as described in 7.6.2.1.

All functions with associated constraints that are not satisfied (13.4.2) 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 13.4.4. 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 13.6.6.2. 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 (*rfd)(double) = f; // selects f(double)
void g() {
   (int (*)(int))&f; // cast expression as selector
}
```

The initialization of \( pfe \) is ill-formed because no \( f() \) with type \( \text{int}(...) \) has been declared, and not because of any ambiguity. For another example,

```c
struct X {
   int f(int);
   static int f(long);
};
```
int (X::*p1)(int) = &X::f;  // OK
int (*p2)(int) = &X::f;     // error: mismatch
int (*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

[Note: If f() and g() are both overloaded functions, the cross product of possibilities must be considered to resolve f(&g), or the equivalent expression f(g). — end note]

[Note: Even if B is a public base of D, we have
D* f();
B* (*p1)() = &f;  // error
void g(D*);
void (*p2)(B*) = &g; // error
—end note]

12.5 Overloaded operators

A function declaration having one of the following operator-function-ids as its name declares an operator function. A function template declaration having one of the following operator-function-ids as its name declares an operator function template. A specialization of an operator function template is also an operator function. An operator function is said to implement the operator named in its operator-function-id.

operator-function-id:
  operator operator

operator: one of

new   delete  new[]  delete[]  co_await ()  [ ]  ->  -*
~     !     +     -     *     /     %     ^     &
|     =     +=    -=    *=    /=    %=    ^=    &=
|=     ==    !=    <     >     <=    >=    <=>   &&
||     <<    >>    <<=   >>=   ++    --    ,

[Note: The operators new[], delete[], (), and [] 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]

Both the unary and binary forms of
+     -     *     &
can be overloaded.

The following operators cannot be overloaded:

.     .*     ::     ?:

nor can the preprocessing symbols # and ## (Clause 15).

Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (12.5.1 – 12.5.7). They can be explicitly called, however, using the operator-function-id as the name of the function in the function call syntax (7.6.1.2). [Example:

complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
—end example]

The allocation and deallocation functions, operator new, operator new[], operator delete and operator delete[], are described completely in 6.6.5.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 6.6.5.4.

The 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
It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators \( = \), \( \& \), and \( , \) (comma), predefined for each type, can be changed for specific class and enumeration types by defining operator functions that implement these operators. Operator functions are inherited in the same manner as other base class functions.

8 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.

9 An operator function cannot have default arguments (9.2.3.6), 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.

10 Operators not mentioned explicitly in subclauses 12.5.3 through 12.5.7 act as ordinary unary and binary operators obeying the rules of 12.5.1 or 12.5.2.

### 12.5.1 Unary operators

A prefix unary operator shall be implemented by a non-static member function (11.3.1) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator \( @ \), \( @x \) can be interpreted as either \( x.\text{operator}() \) or \( \text{operator @} (x) \). If both forms of the operator function have been declared, the rules in 12.3.1.2 determine which, if any, interpretation is used. See 12.5.7 for an explanation of the postfix unary operators ++ and --.

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]

### 12.5.2 Binary operators

A binary operator shall be implemented either by a non-static member function (11.3.1) with one parameter or by a non-member function with two parameters. Thus, for any binary operator \( @ \), \( x@y \) can be interpreted as either \( x.\text{operator}@(y) \) or \( \text{operator @} (x,y) \). If both forms of the operator function have been declared, the rules in 12.3.1.2 determine which, if any, interpretation is used.

### 12.5.3 Assignment

An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator \( \text{operator=} \) is implicitly declared for a class if not declared by the user (11.3.5), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

Any assignment operator, even the copy and move assignment operators, can be virtual. [Note: 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:

```
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]
12.5.4 Function call

operator() shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

\[
\text{postfix-expression } ( \text{expression-list}_{\text{opt}} )
\]

where the postfix-expression evaluates to a class object and the possibly empty expression-list matches the parameter list of an operator() member function of the class. Thus, a call \(x(\text{arg1}, \ldots)\) is interpreted as \(x.\text{operator}()\(\text{arg1}, \ldots\) for a class object \(x\) of type \(T\) if \(T::\text{operator}()\(T1, T2, T3)\) exists and if the operator is selected as the best match function by the overload resolution mechanism (12.3.3).

12.5.5 Subscripting

operator[] shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

\[
\text{postfix-expression } [ \text{expr-or-braced-init-list} ]
\]

Thus, a subscripting expression \(x[y]\) is interpreted as \(x.\text{operator}[]\(y)\ for a class object \(x\) of type \(T\) if \(T::\text{operator}[]\(T1)\) exists and if the operator is selected as the best match function by the overload resolution mechanism (12.3.3). [Example:

```cpp
struct X {
    Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7; // OK: meaning x.\text{operator}[](\{1,2,3\})
int a[10];
a[{1,2,3}] = 7; // error: built-in subscript operator
```

—end example]

12.5.6 Class member access

operator-> shall be a non-static member function taking no parameters. It implements the class member access syntax that uses ->.

\[
\text{postfix-expression } \to \text{template}_{\text{opt}} \ \text{id-expression}
\]

An expression \(x->m\) is interpreted as \((x.\text{operator}->())->m\) for a class object \(x\) of type \(T\) if \(T::\text{operator}->()\) exists and if the operator is selected as the best match function by the overload resolution mechanism (12.3).

12.5.7 Increment and decrement

The user-defined function called operator++ implements the prefix and postfix ++ 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.\[131\]

[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.\text{operator}++()
    a++; // a.\text{operator}++(0)
    ++b; // operator++(b)
    b++; // operator++(b, 0)
}
```

131 Calling operator++ explicitly, as in expressions like a.\text{operator}++(2), has no special properties: The argument to operator++ is 2.
The prefix and postfix decrement operators -- are handled analogously.

### 12.5.8 User-defined literals

**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 (11.8.3)), an explicit instantiation or specialization of a function template, or a `using-declaration` (9.8). 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:

```c
const char*
unsigned long long int
long double
char
wchar_t
char8_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char8_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

If a parameter has a default argument (9.2.3.6), 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 (13.6.3) 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`, they may have internal, module, or external linkage, they can be called explicitly, their addresses can be taken, etc. — end note]

8. [Example:

```c
void operator "\_km(long double); // OK
string operator "\_i18n(const char*, std::size_t); // OK
template <char...> double operator "\u03C0(); // OK: UCN for lowercase pi
float operator "\_e(const char*); // OK
```
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 12.3.1.2 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 12.3.1.2, 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). Similarly, the term promoted arithmetic type refers to floating-point types plus promoted integral types. [Note: In all cases where a promoted integral type or promoted arithmetic type is required, an operand of enumeration type will be acceptable by way of the integral promotions. —end note]

In the remainder of this subclause, vq represents either volatile or no cv-qualifier.

For every pair \((T,\ vq)\), where \(T\) is an arithmetic type other than bool, there exist candidate operator functions of the form

\[vq\ T&\ operator++(vq\ T&);\]
\[T\ operator++(vq\ T&,\ int);\]

For every pair \((T,\ vq)\), where \(T\) is an arithmetic type other than bool, there exist candidate operator functions of the form

\[vq\ T&\ operator--(vq\ T&);\]
\[T\ operator--(vq\ T&,\ int);\]

For every pair \((T,\ vq)\), where \(T\) is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form

\[T*\ vq&\ operator++(T*\ vq&);\]
\[T*\ vq&\ operator--(T*\ vq&);\]
\[T*\ operator++(T*\ vq&\,\ int);\]
\[T*\ operator--(T*\ vq&\,\ int);\]

For every cv-qualified or cv-unqualified object type \(T\), there exist candidate operator functions of the form

\[T&\ operator*(T*);\]

For every function type \(T\) that does not have cv-qualifiers or a ref-qualifier, there exist candidate operator functions of the form

\[T&\ operator*(T*);\]

For every type \(T\) there exist candidate operator functions of the form

\[T*\ operator*(T*);\]

For every promoted arithmetic type \(T\), there exist candidate operator functions of the form

\[T\ operator+(T);\]
\[T\ operator-(T);\]
For every promoted integral type $T$, there exist candidate operator functions of the form

\[ T \text{ operator-}(); \]

For every quintuple $(C_1, C_2, T, cv_1, cv_2)$, where $C_2$ is a class type, $C_1$ is the same type as $C_2$ or is a derived class of $C_2$, and $T$ is an object type or a function type, there exist candidate operator functions of the form

\[ cv_2 \text{ } T& \text{ operator-}*>(cv_1 \text{ } C_1*, \text{ } cv_2 \text{ } T \text{ } C_2::*); \]

where $cv_2$ is the union of $cv_1$ and $cv_2$. The return type is shown for exposition only; see 7.6.4 for the determination of the operator's result type.

For every pair of promoted arithmetic types $L$ and $R$, there exist candidate operator functions of the form

\[
\begin{align*}
L & \text{ operator}\ast(L, R) ; \\
L & \text{ operator}/(L, R) ; \\
L & \text{ operator}+(L, R) ; \\
L & \text{ operator}-(L, R) ; \\
bool & \text{ operator}==(L, R) ; \\
bool & \text{ operator}!=(L, R) ; \\
bool & \text{ operator}<(L, R) ; \\
bool & \text{ operator}>(L, R) ; \\
bool & \text{ operator}<=>(L, R) ;
\end{align*}
\]

where $LR$ is the result of the usual arithmetic conversions (7.4) between types $L$ and $R$.

For every integral type $T$ there exists a candidate operator function of the form

\[ \text{std::strong\_ordering} \text{ operator}==>(T, T); \]

For every pair of floating-point types $L$ and $R$, there exists a candidate operator function of the form

\[ \text{std::partial\_ordering} \text{ operator}==>(L, R); \]

For every cv-qualified or cv-unqualified object type $T$ there exist candidate operator functions of the form

\[
\begin{align*}
T* & \text{ operator}+(T*, \text{ std::ptrdiff\_t}); \\
T& & \text{ operator}[]\text{ } (T*, \text{ std::ptrdiff\_t}); \\
T* & \text{ operator}-(T*, \text{ std::ptrdiff\_t}); \\
T* & \text{ operator}+(\text{ std::ptrdiff\_t}, \text{ } T*); \\
T& & \text{ operator}[]\text{ } (\text{ std::ptrdiff\_t}, \text{ } T*);
\end{align*}
\]

For every $T$, where $T$ is a pointer to object type, there exist candidate operator functions of the form

\[ \text{std::ptrdiff\_t} \text{ operator}-(T, T); \]

For every $T$, where $T$ is an enumeration type or a pointer type, there exist candidate operator functions of the form

\[
\begin{align*}
bool & \text{ operator}==(T, T); \\
bool & \text{ operator}!=(T, T); \\
bool & \text{ operator}<(T, T); \\
bool & \text{ operator}>(T, T); \\
bool & \text{ operator}<=>(T, T); \\
R & \text{ operator}==>(T, T);
\end{align*}
\]

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

\[
\begin{align*}
bool & \text{ operator}==(T, T); \\
bool & \text{ operator}!=(T, T); \\
\text{std::strong\_equality} \text{ operator}==>(T, T);
\end{align*}
\]

For every pair of promoted integral types $L$ and $R$, there exist candidate operator functions of the form

\[
\begin{align*}
LR & \text{ operator}\%\%(L, R); \\
LR & \text{ operator}%(L, R); \\
LR & \text{ operator}^\ast(L, R); \\
LR & \text{ operator}!(L, R); \\
L & \text{ operator}<<>(L, R); \\
L & \text{ operator}>>>(L, R);
\end{align*}
\]

§ 12.6
where \( LR \) is the result of the usual arithmetic conversions (7.4) between types \( L \) and \( R \).

21 For every triple \((L, vq, R)\), where \( L \) is an arithmetic type, and \( R \) is a promoted arithmetic type, there exist candidate operator functions of the form

\[
\begin{align*}
vq \text{ } Lk & \quad \text{operator=}=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}**=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}/=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}+=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}-=(vq \text{ } Lk, R);
\end{align*}
\]

22 For every pair \((T, vq)\), where \( T \) is any type, there exist candidate operator functions of the form

\[
\begin{align*}
T*\text{ } vq & \quad \text{operator=}=(T*\text{ } vq\text{ }, T); \\
\end{align*}
\]

23 For every pair \((T, vq)\), where \( T \) is an enumeration or pointer-to-member type, there exist candidate operator functions of the form

\[
\begin{align*}
vq \text{ } Tk & \quad \text{operator=}=(vq \text{ } Tk, T);
\end{align*}
\]

24 For every pair \((T, vq)\), where \( T \) is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form

\[
\begin{align*}
T*\text{ } vq & \quad \text{operator}+=(T*\text{ } vq\text{ }, \text{std::ptrdiff}\_t); \\
T*\text{ } vq & \quad \text{operator}-=(T*\text{ } vq\text{ }, \text{std::ptrdiff}\_t);
\end{align*}
\]

25 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

\[
\begin{align*}
vq \text{ } Lk & \quad \text{operator}%=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}<<=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}>>=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}&=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}^=(vq \text{ } Lk, R); \\
vq \text{ } Lk & \quad \text{operator}|=(vq \text{ } Lk, R);
\end{align*}
\]

26 There also exist candidate operator functions of the form

\[
\begin{align*}
\text{bool } & \quad \text{operator}!(\text{bool}); \\
\text{bool } & \quad \text{operator}&&(\text{bool}, \text{bool}); \\
\text{bool } & \quad \text{operator}||(\text{bool}, \text{bool});
\end{align*}
\]

27 For every pair of promoted arithmetic types \( L \) and \( R \), there exist candidate operator functions of the form

\[
\begin{align*}
LR & \quad \text{operator}?:=(\text{bool}, L, R);
\end{align*}
\]

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]

28 For every type \( T \), where \( T \) is a pointer, pointer-to-member, or scoped enumeration type, there exist candidate operator functions of the form

\[
\begin{align*}
T & \quad \text{operator}?:=(\text{bool}, T, T);
\end{align*}
\]
13 Templates

1 A template defines a family of classes, functions, or variables, an alias for a family of types, or a concept.

\[ \text{template-declaration:} \]
\[ \text{template-head declaration} \]
\[ \text{template-head concept-definition} \]

\[ \text{template-head:} \]
\[ \text{template} < \text{template-parameter-list} > \text{requires-clause}_{\text{opt}} \]

\[ \text{template-parameter-list:} \]
\[ \text{template-parameter} \]
\[ \text{template-parameter-list} \text{ , } \text{template-parameter} \]

\[ \text{requires-clause:} \]
\[ \text{requires constraint-logical-or-expression} \]

\[ \text{constraint-logical-or-expression:} \]
\[ \text{constraint-logical-and-expression} \]
\[ \text{constraint-logical-or-expression} \text{ || constraint-logical-and-expression} \]

\[ \text{constraint-logical-and-expression:} \]
\[ \text{primary-expression} \]
\[ \text{constraint-logical-and-expression} \text{ && primary-expression} \]

\[ \text{concept-definition:} \]
\[ \text{concept concept-name} = \text{constraint-expression} ; \]

\[ \text{concept-name:} \]
\[ \text{identifier} \]

\[ \text{type-constraint:} \]
\[ \text{nested-name-specifier}_{\text{opt}} \text{ concept-name} \]
\[ \text{nested-name-specifier}_{\text{opt}} \text{ concept-name} < \text{template-argument-list}_{\text{opt}} > \]

[Note: The > token following the template-parameter-list of a template-declaration may be the product of replacing a >> token by two consecutive > tokens (13.2). — end note]

2 The declaration in a template-declaration (if any) shall
   (2.1) declare or define a function, a class, or a variable, or
   (2.2) define a member function, a member class, a member enumeration, or a static data member of a class
         template or of a class nested within a class template, or
   (2.3) define a member template of a class or class template, or
   (2.4) be a deduction-guide, or
   (2.5) be an alias-declaration.

3 A template-declaration is a declaration. A template-declaration is also a definition if its template-head is
   followed by either a concept-definition or a declaration that defines a function, a class, a variable, or a static
   data member. 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:]
\[
\begin{align*}
\text{template<class T> } \\
\text{constexpr T pi} & = \text{T(3.1415926535897932385L)}; \\
\text{template<class T> } \\
\text{T circular_area(T r) } \\
\{} \\
\text{ \quad return pi<T> } \ast \text{ r } \ast \text{ r; } \\
\{} \\
\text{struct matrix_constants } \\
\text{\{} \\
\text{\quad template<class T> } \\
\text{\quad using pauli = hermitian_matrix<T, 2>; } \\
\text{\quad template<class T> } \\
\text{\quad constexpr static pauli<T> sigma = } \{ \{ 0, 1 \}, \{ 1, 0 \} \}; \\
\end{align*}
\]
A template-declaration can appear only as a namespace scope or class scope declaration. Its declaration shall not be an export-declaration. In a function template declaration, the last component of the declarator-id shall not be a template-id. [Note: 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 (13.6.5). —end note]

In a template-declaration, explicit specialization, or explicit instantiation the init-declarator-list in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.

A 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 \( C<\!\!T\!\!, \ldots, \!\!A_i\!\!, \ldots, \!\!A_n> \), then let \( E' \) be \( C<T, A_1, \ldots, A_n> \). Otherwise, let \( E' \) be \( C<T> \). If \( T \) is not a pack, then \( E \) is \( E' \), otherwise \( E \) is \( (E' \&\& \ldots) \). This constraint-expression \( E \) is called the immediately-declared constraint of \( T \). The concept designated by a type-constraint shall be a type concept (13.6.8).

A template name has linkage (6.5). 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 (13.8.3), 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 (6.2). [Note: Default arguments for function templates and for member functions of class templates are considered definitions for the purpose of template instantiation (13.6) and must also obey the one-definition rule. —end note]

A class template shall not have the same name as any other template, class, function, variable, enumeration, enumerator, namespace, or type in the same scope (6.3), except as specified in 13.6.5. Except that a function template can be overloaded either by non-template functions (9.2.3.5) with the same name or by other function templates with the same name (13.9.3), a template name declared in namespace scope or in class scope shall be unique in that scope.

A templated entity is

- a template,
- an entity defined (6.1) or created (6.6.7) in a templated entity,
- a member of a templated entity,
- an enumerator for an enumeration that is a templated entity, or
- 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]

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 (13.4.2) on template arguments (13.3). 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 (Clause 7). [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]

A function template, member function of a class template, variable template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated (13.8.1) unless the corresponding specialization is explicitly instantiated (13.8.2) in some translation unit; no diagnostic is required.
13.1 Template parameters

1. The syntax for template-parameters is:

   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

   [Note: The > token following the template-parameter-list of a type-parameter may be the product of replacing a >> token by two consecutive > tokens (13.2). — 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:

   ```
   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,

   ```
   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 non-type template-parameter shall have one of the following (optionally cv-qualified) types:

   (4.1) — a literal type that has strong structural equality (11.10.1),
   (4.2) — an lvalue reference type,
   (4.3) — a type that contains a placeholder type (9.1.7.5), or
   (4.4) — a placeholder for a deduced class type (9.1.7.6).

   [Note: Other types are disallowed either explicitly below or implicitly by the rules governing the form of template-arguments (13.3). — end note] The top-level cv-qualifiers on the template-parameter are ignored when determining its type.

5. When a non-type template-parameter of non-reference and non-class type is used as an initializer for a reference, a temporary is always used. An id-expression naming a non-type template-parameter of class

---

132) 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.
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. [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). — end note] [Example:

```c
struct A { friend bool operator==(const A&, const A&) = default; }

template< const X& x, int i, A a > void f() {
    i++;
    // error: change of template-parameter value
    &x;
    // OK
    &i;
    // error: address of non-reference template-parameter
    &a;
    // OK
    int& ri = i;
    // error: non-const reference bound to temporary
    const int& cri = i;
    // OK: const reference bound to temporary
    const A& ra = a;
    // OK: const reference bound to a template parameter object
}
```

—end example] 7

A non-type template-parameter shall not be declared to have floating-point or void type. [Example:

```c
template<double d> class X;
// error

template<double* pd> class Y;
// OK

template<double& rd> class Z;
// OK
```

—end example] 8

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:

```c
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];
// OK due to implicit argument conversion

R<>& y;
// OK due to both adjustment and conversion

S<>& z;
```

—end example] 9

A type-parameter that starts with a type-constraint introduces the immediately-declared constraint (Clause 13) of the parameter. [Example:

```c
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] 10

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 invented type corresponding to the placeholder (9.2.3.5).

11 A default template-argument is a template-argument (13.3) 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 (13.6.3). 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 (9.2.3.6). [Example:

```
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
```

is equivalent to

```
template<class T1 = int, class T2 = int> class A;
```

—end example]

13 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 (9.2.3.5) of the function template or has a default argument (13.9.2). A template parameter of a deduction guide template (13.10) that does not have a default argument shall be deducible from the parameter-type-list of the deduction guide template. [Example:

```
template<class T1 = int, class T2> class B;
// error

// U can be neither deduced from the parameter-type-list nor specified
template<class... T, class... U> void f() { } // error
template<class... T, class U> void g() { } // error
```

—end example]

14 A template-parameter shall not be given default arguments by two different declarations in the same scope. [Example:

```
template<class T = int> class X;
template<class T = int> class X { /* ... */ }; // error
```

—end example]

15 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:

```
template<int i = 3 > 4 > // syntax error
class X { /* ... */ };
```

```
template<int i = (3 > 4) > // OK
class Y { /* ... */ };
```

—end example]

16 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:

```
template <class T = float> struct B {};
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]

17 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 (9.2.3.5), then the template-parameter is a template parameter pack (13.6.3). 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
               // but not a pack expansion

template <class T, int... Dims>
    struct multi_array; // Dims is a non-type template parameter pack
                        // but not a pack expansion

template <class... T>
    struct value_holder {
        template <T... Values> struct apply { }; // Values is a non-type template parameter pack
    };
                        // and a pack expansion

template <class... T, T... Values>
    struct static_array; // error: Values expands template type parameter
                        // pack T within the same template parameter list
```
—end example]

13.2 Names of template specializations [temp.names]

A template specialization (13.8) 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 ... opt
    template-argument-list, template-argument ... opt

template-argument:
    constant-expression
    type-id
    id-expression
```

[Note: The name lookup rules (6.4) 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 (6.4.2). — 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 > 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:

```cpp
    template<int i> class X { /* ... */ };
```

```
X< 1>2 > x1; // syntax error
X<(1>2)> x2; // OK
```
The keyword \texttt{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 (Clause 11) or enum-head-name (9.6), the keyword \texttt{template} shall not appear at the top level. In a qualified-id used as the name in a typename-specifier (13.7), elaborated-type-specifier (9.1.7.3), using-declaration (9.8), or class-or-decltype (11.6), an optional keyword \texttt{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 (13.7.2.1), the member template name shall be prefixed by the keyword \texttt{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->template alloc<200>(); // ill-formed: < means less than
    T* p2 = p->template alloc<200>(); // OK: < starts template argument list
    T::adjust<100>(); // ill-formed: < means less than
    T::template adjust<100>(); // OK: < starts template argument list
}
```

\textit{—end example}\]

A name prefixed by the keyword \texttt{template} shall be a \texttt{template-id} or the name shall refer to a class template or an alias template. [Note: The keyword \texttt{template} may not be applied to non-template members of class templates. \textit{—end note}] [Note: As is the case with the \texttt{typename} prefix, the \texttt{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 \texttt{->} or \texttt{.} is not dependent on a template-parameter, or the use does not appear in the scope of a template. \textit{—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
}
```

\textit{—end example}\]

[Note: A simple-template-id that names a class template specialization is a class-name (11.2). Any other simple-template-id that names a type is a typedef-name. \textit{—end note}] 

A template-id is valid if

\begin{enumerate}
\item there are at most as many arguments as there are parameters or a parameter is a template parameter pack (13.6.3),
\item there is an argument for each non-deducible non-pack parameter that does not have a default template-argument,
\item each template-argument matches the corresponding template-parameter (13.3),
\end{enumerate}
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 (13.9.2). [Example:

\[
\begin{align*}
\text{template<class } & \text{T, T::type n = 0> class X;} \\
\text{struct S} & \{ \\
& \text{ using type = int; }; \\
& \text{ using T1 = X<S, int, int>; } & \text{// error: too many arguments} \\
& \text{ using T2 = X<>; } & \text{// error: no default argument for first template parameter} \\
& \text{ using T3 = X<int>;} & \text{// error: value 1 does not match type-parameter} \\
& \text{ using T4 = X<int>; } & \text{// error: substitution failure for second template parameter} \\
& \text{ using T5 = X<S>;} & \text{// OK} \\
\end{align*}
\]

—end example]

8 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
(13.7), and all template-arguments in the simple-template-id are non-dependent (13.7.2.4), the associated
constraints (13.4.2) of the constrained template shall be satisfied (13.4.1). [Example:

\[
\begin{align*}
\text{template<typename } & \text{T> concept C1 = sizeof(T) != sizeof(int);} \\
\text{template<} & \text{C1 T> struct S1 } \{ \}; \\
\text{template<} & \text{C1 T> using Ptr = T*;} \\
\text{S1<int> } & \text{* p;} & \text{// error: constraints not satisfied} \\
\text{Ptr<int> } & \text{* p;} & \text{// error: constraints not satisfied} \\
\text{template<typename } & \text{T>} \\
\text{struct S2 } \{ \text{Ptr<int> x;} \}; & \text{// error, no diagnostic required} \\
\text{template<typename } & \text{T>} \\
\text{struct S3 } \{ \text{Ptr<T> x;} \}; & \text{// OK, satisfaction is not required} \\
\text{S3<int> } & \text{x;} & \text{// error: constraints not satisfied} \\
\text{template<typename } & \text{T> class X>} \\
\text{struct S4 } \{ \\
& \text{X<int> x;} & \text{// error, no diagnostic required} \\
\}; \\
\text{template<typename } & \text{T> concept C2 = sizeof(T) == 1;} \\
\text{template<} & \text{C2 T> struct S } \{ \}; \\
\text{template struct S<} & \text{char[2]>;} & \text{// error: constraints not satisfied} \\
\text{template<> struct S<} & \text{char[2]> } \{ \}; & \text{// error: constraints not satisfied} \\
\end{align*}
\]

—end example]

13.3 Template arguments

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 (13.6.3), it will
correspond to zero or more template-arguments. [Example:

\[
\begin{align*}
\text{template<class } & \text{T> class Array } \{ \\
& \text{T* v;} \\
& \text{int sz;} \\
& \text{public:} \\
& \text{explicit Array(int);} \\
& \text{T& operator[] (int);} \\
\end{align*}
\]

§ 13.3
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.\(^{134}\) [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 { /* ... */ }; // S is accessible
    X<S> x; // OK: S is accessible
    // X<Y::S> has a static member of type Y::S
    // OK: even though Y::S is private
  };

  X<Y::S> y; // error: S not accessible
—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 { /* ... */ }; // S is accessible
  };

  A<S> b; // ill-formed: 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:

```cpp
template<class T = char> class String;
String<>* p; // OK: String<char>
```

\(^{134}\) There is no such ambiguity in a default template-argument because the form of the template-parameter determines the allowable forms of the template-argument.
An explicit destructor call (11.3.6) for an object that has a type that is a class template specialization may explicitly specify the template-arguments. [Example:

```cpp
template<class T> struct A {
    ~A();
};
void f(A<int>*, A<int>*) { 
    p->A<int>::~A(); // OK: destructor call
    q->A<int>::~A(); // OK: destructor call
}
```
—end example]

5 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 (13.8.1) when the value of that default argument is needed. [Example:

```cpp
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 (13.6.3).

### 13.3.1 Template type arguments

1 A template-argument for a template-parameter which is a type shall be a type-id.

2 [Example:

```cpp
template <class T> class I { };
template <class T> void f(T t) { }
struct { } unnamed_obj;

void f() {
    struct A { }; // OK
enum { e1 }; // OK
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 (6.7). —end note]

### 13.3.2 Template non-type arguments

1 If the type T of a template-parameter (13.1) contains a placeholder type (9.1.7.5) or a placeholder for a deduced class type (9.1.7.6), the type of the parameter is the type deduced for the variable x in the invented declaration

```
T x = template-argument ;
```
If a deduced parameter type is not permitted for a template-parameter declaration (13.1), 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. 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 subobject (6.6.2),
2. a temporary object (6.6.7),
3. a string literal (5.13.5),
4. the result of a typeid expression (7.6.1.7), or
5. a predefined __func__ variable (9.4.1).

[Note: If the template-argument represents a set of overloaded functions (or a pointer or member pointer to such), the matching function is selected from the set (12.4). — end note]

[Example:

```cpp
template<const int* pci> struct X { /* ... */ };  
int ai[10];  
X<ai> xi;   // array to pointer and qualification conversions

struct Y { /* ... */ };  
template<const Y& b> struct Z { /* ... */ };  
Y y;  
Z<y> z;   // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W { /* ... */ };  
int b[5];  
W<b> w;   // no conversion

void f(char);  
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };  
A<&f> a;  // selects f(int)

template<auto n> struct B { /* ... */ };  
B<5> b1;   // OK: template parameter type is int  
B<"a"> b2;  // OK: template parameter type is char  
B<2.5> b3;  // error: template parameter type cannot be double
```

—end example

—end note]

[Note: A string literal (5.13.5) is not an acceptable template-argument for a template-parameter of non-class type. [Example:

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

X<const char*, "Studebaker"> x; // error: string literal as template-argument  

const char p[] = "Vivisectionist";  
X<const char*, p> y;  // OK

struct A {  
   constexpr A(const char*) {}
};  
friend bool operator==(const A&, const A&) = default;

X<A, "Pyrophoricity"> z;  // OK, string literal is a constructor argument to A

—end example] — end note]
Note: The address of an array element or non-static data member is not an acceptable template-argument.  

Example:

```cpp
template<int* p> class X { }

int a[10];
struct S { int m; static int s; } s;

X<&a[2]> x3; // error: address of array element
X<&a.m> x4; // error: address of non-static member
X<&S.s> x5; // OK: address of static member
X<&S::s> x6; // OK: address of static member
```

—end example] —end note]

Note: A temporary object is not an acceptable template-argument when the corresponding template-parameter has reference type.  

Example:

```cpp
template<const int& CRI> struct B { /* ... */ }

B<1> b2; // error: temporary would be required for template argument
int c = 1;
B<c> b1; // OK
```

—end example] —end note]

### 13.3.3 Template template arguments

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 (13.6.5) 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:

```cpp
template<class T> class A { /* ... */ }

template<class T, class U = T> class B { /* ... */ }
```

A template-argument matches a template template-parameter \( P \) when \( P \) is at least as specialized as the template-argument \( A \). 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 (13.6.6.1), and for template template-parameters, each of their corresponding template-parameters matches, recursively. When \( P \)’s template-head contains a template parameter pack (13.6.3), the template parameter pack will match zero or more template parameters or template parameter packs in the template-head of \( A \) with the same type and form as the template parameter pack in \( P \) (ignoring whether those template parameters are template parameter packs).

Example:

```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> class P> class X { /* ... */ };  
template<template<class ...> class Q> 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:

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:

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<templateโอกาส Template parameter P apat 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 (13.6.6.2). 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.
(4.2) — Each function template has a single function parameter whose type is a specialization of X with template arguments corresponding to the template parameters from the respective function template where, for each template parameter PP in the template-head of the function template, a corresponding template argument AA is formed. If PP declares a template parameter pack, then AA is the id-expression PP.

If the rewrite produces an invalid type, then P is not at least as specialized as A.
13.4 Template constraints

1 [Note: This subclause defines the meaning of constraints on template arguments. The abstract syntax and satisfaction rules are defined in 13.4.1. Constraints are associated with declarations in 13.4.2. Declarations are partially ordered by their associated constraints (13.4.4). — end note]

13.4.1 Constraints

1 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:

(1.1) — conjunctions (13.4.1.1),
(1.2) — disjunctions (13.4.1.1), and
(1.3) — atomic constraints (13.4.1.2)

2 In order for a constrained template to be instantiated (13.8), its associated constraints (13.4.2) 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 (13.2) requires the satisfaction of its constraints. Overload resolution (12.3.2) requires the satisfaction of constraints on functions and function templates. — end note]

13.4.1.1 Logical operations

1 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 ∧ and disjunction is spelled using the symbol ∨. The operands of these operations are called the left and right operands. In the constraint A ∧ B, A is the left operand, and B is the right operand. — end note]

2 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.

3 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 conjunction is not satisfied. Otherwise, the disjunction is satisfied if and only if the second operand is satisfied.

4 [Example:]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 ∧ get_value<T>()

    void f(int);

    f('a'); // OK: calls f(int)
In the satisfaction of the associated constraints (13.4.2) of f, the constraint sizeof(char) > 1 is not satisfied; the second operand is not checked for satisfaction. — end example]

13.4.1.2 Atomic constraints

1 An atomic constraint is formed from an expression E and a mapping from the template parameters that appear within E to template arguments involving the template parameters of the constrained entity, called the parameter mapping (13.4.2). [Note: Atomic constraints are formed by constraint normalization (13.4.3). E is never a logical AND expression (7.6.14) nor a logical OR expression (7.6.15). — end note]

2 Two atomic constraints are identical if they are formed from the same expression and the targets of the parameter mappings are equivalent according to the rules for expressions described in 13.6.6.1.

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. [Example:template<typename T> concept C =
    sizeof(T) == 4 && !true; // requires atomic constraints sizeof(T) == 4 and !true

§ 13.4.1.2

348
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

13.4.2 Constrained declarations

A template declaration (Clause 13) or function declaration (9.2.3.5) can be constrained by the use of a requires-clause. This allows the specification of constraints for that declaration as an expression:

\[
\text{constraint-expression: logical-or-expression}
\]

Constraints can also be associated with a declaration through the use of type-constraints in a template-parameter-list. Each of these forms introduces additional constraint-expressions that are used to constrain the declaration.

A template’s associated constraints are defined as follows:

1. If there are no introduced constraint-expressions, the declaration has no associated constraints.
2. Otherwise, if there is a single introduced constraint-expression, the associated constraints are the normal form (13.4.3) of that expression.
3. Otherwise, the associated constraints are the normal form of a logical AND expression (7.6.14) whose operands are in the following order:
   1. the constraint-expression introduced by each type-constraint (13.1) in the declaration’s template-parameter-list, in order of appearance, and
   2. the constraint-expression introduced by a requires-clause following a template-parameter-list (Clause 13), and
   3. the constraint-expression introduced by each type-constraint in the parameter-type-list of a function declaration, and
   4. the constraint-expression introduced by a trailing requires-clause (9.2) of a function declaration (9.2.3.5).

The formation of the associated constraints establishes the order in which constraints are instantiated when checking for satisfaction (13.4.1). [Example:

\[
\text{template<typename T> concept C = true;}
\text{template<typename T> void f1(T);}
\text{template<typename T> void C<T> void f2(T);}
\text{template<typename T> void f3(T) requires C<T>;}
\]

The functions f1, f2, and f3 have the associated constraint C<T>.

\[
\text{template<typename T> concept C1 = true;}
\text{template<typename T> concept C2 = sizeof(T) > 0;}
\text{template<C1 T> void f4(T) requires C2<T>;}
\text{template<typename T> requires C1<T> && C2<T> void f5(T);}
\]

The associated constraints of f4 and f5 are C1<T> \&\& C2<T>.

\[
\text{template<C1 T> requires C2<T> void f6();}
\text{template<C2 T> requires C1<T> void f7();}
\]

The associated constraints of f6 are C1<T> \&\& C2<T>, and those of f7 are C2<T> \&\& C1<T>. — end example]
13.4.3 Constraint normalization

The normal form of an expression $E$ is a constraint (13.4.1) that is defined as follows:

1. The normal form of an expression $E$ is the normal form of $E$.
2. The normal form of an expression $E_1 \mid\mid E_2$ is the disjunction (13.4.1.1) of the normal forms of $E_1$ and $E_2$.
3. The normal form of an expression $E_1 \& \& E_2$ is the conjunction of the normal forms of $E_1$ and $E_2$.
4. The normal form of an id-expression of the form $C<A_1, A_2, \ldots, A_n>$, where $C$ names a concept, is the normal form of the constraint-expression of $C$, after substituting $A_1, A_2, \ldots, A_n$ 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*>;  // #1
template<typename V> concept C = B<V&>;  // #2
```

Normalization of B’s constraint-expression is valid and results in \(T::value\) (with the mapping \(T \mapsto U\)) \(\lor\) \(true\) (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 \(T\&\ast\) in the parameter mapping. — end example]

2. The normal form of any other expression $E$ is the atomic constraint whose expression is $E$ and whose parameter mapping is the identity mapping.

3. 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 (13.4.1) of a declaration and when evaluating the value of an id-expression that names a concept specialization (7.5.4). — end note]

[Example:

```
template<typename T> concept C1 = sizeof(T) == 1;
template<typename T> concept C2 = C1<T>() \&\& 1 == 2;
template<typename T> concept C3 = requires { typename T::type; };
template<typename T> concept C4 = requires (T x) { ++x; }
```

The associated constraints of #1 are \(\text{sizeof}(T) == 1\) (with mapping \(T \mapsto U\)) \(\land\) \(1 == 2\).
The associated constraints of #2 are \(\text{requires}\{\text{typename } T::\text{type};\}\) (with mapping \(T \mapsto U\)).
The associated constraints of #3 are \(\text{requires}\{\text{T } x\} \{\text{++x; }\}\) (with mapping \(T \mapsto U\)). — end example]

13.4.4 Partial ordering by constraints

A constraint $P$ subsumes a constraint $Q$ if and only if, for every disjunctive clause $P_i$ in the disjunctive normal form 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 the $A$ and $B$ are identical using the rules described in 13.4.1.2.

[Example: Let $A$ and $B$ be atomic constraints (13.4.1.2). 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

135] A constraint is in disjunctive normal form when it is a disjunction of clauses where each clause is a conjunction of atomic constraints. [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 disjunctive clauses are \((A \land B)\) and \((A \land C)\). — end example]

136] A constraint is in conjunctive normal form when it is a conjunction of clauses where each clause is a disjunction 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 $A$ and $(B \lor C)$. — end example]
— the best viable candidate of non-template functions (12.3.3),
— the address of a non-template function (12.4),
— the matching of template template arguments (13.3.3),
— the partial ordering of class template specializations (13.6.5.2), and
— the partial ordering of function templates (13.6.6.2).

— end note]

3 A declaration $D_1$ is at least as constrained as a declaration $D_2$ if

(3.1) $D_1$ and $D_2$ are both constrained declarations and $D_1$’s associated constraints subsume those of $D_2$; or

(3.2) $D_2$ has no associated constraints.

4 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:

```cpp
template<typename T> concept C1 = requires(T t) { --t; }; 
template<typename T> concept C2 = C1<T> && requires(T t) { *t; }; 

template<C1 T> void f(T); // #1 
template<C2 T> void f(T); // #2 
template<typename T> void g(T); // #3 
template<C1 T> void g(T); // #4 
```

f(0); // selects #1
f((int*)0); // selects #2
g(true); // selects #3 because C1<bool> is not satisfied
g(0); // selects #4

— end example]

13.5 Type equivalence [temp.type]

1 Two template-ids refer to the same class, function, or variable if

(1.1) their template-names, operator-function-ids, or literal-operator-ids refer to the same template and

(1.2) their corresponding type template-arguments are the same type and

(1.3) their corresponding non-type template-arguments of pointer-to-member type refer to the same class
    member or are both the null member pointer value and

(1.4) their corresponding non-type template-arguments of reference type refer to the same object or function
    and

(1.5) their remaining corresponding non-type template-arguments have the same type and value after
    conversion to the type of the template-parameter, where they are considered to have the same value if
    they compare equal with the == operator (7.6.10), and

(1.6) their corresponding template template-arguments refer to the same template.

[Example:

```cpp
template<class E, int size> class buffer { /* ... */ }; 
buffer<char,2*512> x; 
buffer<char,1024> y; 
declares x and y to be of the same type, and

template<class T, void(*err_fct)()> class list { /* ... */ }; 
list<int,&error_handler1> x1; 
list<int,&error_handler2> x2; 
list<int,&error_handler2> x3; 
list<char,&error_handler2> x4; 
declares x2 and x3 to be of the same type. Their type differs from the types of x1 and x4.
```

§ 13.5 351
X<Z<int> > z;
declares y and z to be of the same type. —end example]

2 If an expression e is type-dependent (13.7.2.2), decltype(e) denotes a unique dependent type. Two such
decay-type-specifiers refer to the same type only if their expressions are equivalent (13.6.6.1). [Note: However,
such a type may be aliased, e.g., by a typedef-name. —end note]

13.6 Template declarations

1 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:

```cpp
template<class T1, class T2, int I> class A<T1, T2, I> { }; // error
```

—end example] [Note: However, this syntax is allowed in class template partial specializations (13.6.5).
—end note]

2 For purposes of name lookup and instantiation, default arguments, type-constraints, requires-clauses (Clause
13), 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 (8.4.1) 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.

13.6.1 Class templates

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:

```cpp
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};
```

The prefix template<class T> specifies that a template is being declared and that a type-name T may be
used in the declaration. In other words, Array is a parameterized type with T as its parameter. —end example]

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 (13.6.6.1).
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:

```cpp
template<class T1, class T2> struct A {
  void f0();
  void f2();
};
```

```cpp
template<class T2, class T1> void A<T2,T1>::f1() { } // OK
```

```cpp
template<class T2, class T1> void A<T1,T2>::f2() { } // error
```

```cpp
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<C T> struct S {
    void f();
    void g();
    void h();
template<D U> struct Inner;
};

template<C A> void S<A>::f() { }  // OK: template-heads match
template<typename T> void S<T>::g() { }  // error: no matching declaration for S<T>

void S<T>::h() { }  // error (no diagnostic required): template-heads are functionally equivalent but not equivalent

template<C X> template<D Y>
struct S<X>::Inner { };  // OK
—end example] 4

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 (9.1.7.3).

13.6.1.1 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) { 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 {
    typename T::type;
};

template<typename T> struct S {
    void f() requires C<T>;
    void g() requires C<T>;
};

template<typename T>
void S<T>::f() requires C<T> { }  // OK
template<typename T>
void S<T>::g() { }  // error: no matching function in S<T>
—end example]

§ 13.6.1.1
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.]

```cpp
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7; // Array<int>::operator[]
v2[3] = dcomplex(7,8); // Array<dcomplex>::operator[]
```

—end example

### 13.6.1.2 Member classes of class templates [temp.mem.class]

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 (13.8.1). For example,

```cpp
template<class T> struct A {
    class B;
};
A<int>::B* b1; // OK: requires `A` to be defined but not `A::B`

template<class T> class A<T>::B { }; // OK: requires `A::B` to be defined
```

—end note

### 13.6.1.3 Static data members of class templates [temp.static]

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:]

```cpp
template<class T> class X {
    static T s;
};
template<class T> T X<T>::s = 0;

struct limits {
    template<class T>
        static const T min; // declaration
};

template<class T>
const T limits::min = { }; // definition
```

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:]

```cpp
template <class T> struct A {
    static int i[];
};
template <class T> int A<T>::i[4]; // 4 elements

template <> int A<int>::i[] = { 1 }; // OK: 1 element
```

—end example

### 13.6.1.4 Enumeration members of class templates [temp.mem.enum]

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 }; // OK
A<int>::E e = A<int>::e1;
```

—end example
13.6.2 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 (13.6.6.1). [Example:

```
template<class T> struct string {
  template<class T2> int compare(const T2& s) {
    /* ... */
  }
};
```
—end example] [Example:

```
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]
```

A local class of non-closure type shall not have member templates. Access control rules (11.8) apply to member template names. A destructor shall not be a member template. A non-template member function (9.2.3.5) 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:

```
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
```

```
int main() {
  A<char> ac;
  ac.f(1);  // non-template
  ac.f('c');  // template
  ac.f<>();  // template
}
—end example]

A member function template shall not be virtual. [Example:

```
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:

```
§ 13.6.2
355
```
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>(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:

```cpp
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
```

— end example] [Note: There is no syntax to form a template-id (13.2) by providing an explicit template argument list (13.9.1) for a conversion function template (11.3.7.2). — 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 (13.9.2.3), 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 (12.3.3.2) and partial ordering (13.6.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

### 13.6.3 Variadic templates [temp.variadic]

1. A template parameter pack is a template parameter that accepts zero or more template arguments. [Example:

```cpp
tuple<T...> t0; // Types contains no arguments
tuple<int> t1; // Types contains one argument: int
tuple<int, float> t2; // Types contains two arguments: int and float
tuple<> error; // error: 0 is not a type
```

— end example]

2. A function parameter pack is a function parameter that accepts zero or more function arguments. [Example:

```cpp
f(); // args contains no arguments
f<>(i); // 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
```
A pack is a template parameter pack, a function parameter pack, or an init-capture pack. The number of elements of a template parameter pack or a function parameter pack is the number of arguments provided for the parameter pack. The number of elements of an init-capture pack is the number of elements in the pack expansion of its initializer.

A pack expansion consists of a pattern and an ellipsis, the instantiation of which produces zero or more instantiations of the pattern in a list (described below). The form of the pattern depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

1. In a function parameter pack (9.2.3.5); the pattern is the parameter-declaration without the ellipsis.
2. In a using-declaration (9.8); the pattern is a using-declarator.
3. In a template parameter pack that is a pack expansion (13.1):
   - if the template parameter pack is a parameter-declaration; the pattern is the parameter-declaration without the ellipsis;
   - if the template parameter pack is a type-parameter; the pattern is the corresponding type-parameter without the ellipsis.
4. In an initializer-list (9.3); the pattern is an initializer-clause.
5. In a base-specifier-list (11.6); the pattern is a base-specifier.
6. In a mem-initializer-list (11.9.2) for a mem-initializer whose mem-initializer-id denotes a base class; the pattern is the mem-initializer.
7. In a template-argument-list (13.3); the pattern is a template-argument.
8. In an attribute-list (9.11.1); the pattern is an attribute.
9. In an alignment-specifier (9.11.2); the pattern is the alignment-specifier without the ellipsis.
10. In a capture-list (7.5.5); the pattern is a capture.
11. In a sizeof... expression (7.6.2.4); the pattern is an identifier.
12. 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);
```

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 {}
```

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.
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) {
    // OK:
    Args is expanded by the function parameter pack args
    f(const_cast<const Args*>(&args)...);
    // OK: "Args" and "args" are expanded
    f(5 ...);
    // error: pattern does not contain any packs
    f(args);
    // error: pack "args" is not expanded
    f(h(args ...) + args ...);
    // OK: first "args" expanded within h,
    // second "args" expanded within f
}

—end example[
8
The instantiation of a pack expansion that is neither a sizeof... expression nor a fold-expression produces a list of elements $E_1, E_2, \ldots, E_N$, where $N$ is the number of elements in the pack expansion parameters. Each $E_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its $i$th element. Such an element, in the context of the instantiation, is interpreted as follows:

(8.1) — if the pack is a template parameter pack, the element is a template parameter (13.1) of the corresponding kind (type or non-type) designating the $i$th corresponding type or value template argument;

(8.2) — 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

(8.3) — 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]

9
The instantiation of a sizeof... expression (7.6.2.4) produces an integral constant containing the number of elements in the pack it expands.

10
The instantiation of a fold-expression produces:

(10.1) — $(E_1 op E_2) op \cdots op E_N$ for a unary left fold,

(10.2) — $E_1 op (\cdots op (E_{N-1} op E_N))$ for a unary right fold,

(10.3) — $((E op E_1) op E_2) op \cdots op E_N$ for a binary left fold, and

(10.4) — $E_1 op (\cdots op (E_{N-1} op (E_N op E)))$ for a binary right fold.

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 15; if the operator is not listed in Table 15, the instantiation is ill-formed.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value when pack is empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>void()</td>
</tr>
</tbody>
</table>

Table 15 — Value of folding empty sequences

13.6.4 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 (13.9.2.6), 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 task<T>* process(task<T>*);
friend task<T>* preempt<T>(task<T>*);

template<class C> friend int func(C);

friend class task<int>;
template<class P> friend class frd;
};
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit template-arguments, each specialization of the `task` class template has an appropriately typed function `process` as a friend, and this friend is not a function template specialization; because the friend `preempt` has an explicit template-argument `T`, each specialization of the `task` class template has the appropriate specialization of the function template `preempt` as a friend; and each specialization of the `task` class template has all specializations of the function template `func` as friends. Similarly, each specialization of the `task` class template has all specializations of the class template specialization `task<int>` as a friend, and has all specializations of the class template `frd` as friends. — end example |

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 (13.8.1), partially specialized (13.6.5) or explicitly specialized (13.8.3), are friends of the class containing the template friend declaration. [Example:

```cpp
class X {
    template<class T> friend struct A;
    class Y { };
};

template<class T> struct A { X::Y ab; };
// OK
template<class T> struct A<T*> { X::Y ab; };
// OK

—end example]

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 \( C \) (13.9.2.5). 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 { }
    void f();
    struct D {
        void g();
    };
    T h();
    template<T U> T i();
};
template<int> struct A<int> {  // grants friendship to A<int>::B even though
    struct B { };
    int f();
    struct D {
        void g();
    };
    template<int U> int i();
};
template<float*> struct A<float*> {  // grants friendship to A<float*>::h()
    int *h();
};

class C {
    template<class T> friend struct A<T>::B;  // grants friendship to A<int>::B even though
    // it is not a specialization of A<T>::B
    template<class T> friend void A<T>::f();  // does not grant friendship to A<int>::f()
    // because its return type does not match
    template<class T> friend void A<T>::D::g();  // ill-formed: 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()
    template<class T> template<T U> friend T A<T>::i();  // grants friendship to instantiations of A<T>::i() and
    // to A<int>::i(), and thereby to all specializations
};

—end example]

[Note: A friend declaration may first declare a member of an enclosing namespace scope (13.7.5). — end note]

A friend template shall not be declared in a local class.

Friend declarations shall not declare partial specializations. [Example:

```cpp
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 shall not have a requires-clause.

13.6.5 Class template partial specializations

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 (13.6.5.1). 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 (13.6.5.3).

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 (Clause 13). [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 (13.4.4). —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 <T1, T2, I>. —end example] [Note: The template argument list shall not be specified in the primary template declaration. For example,

   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 (9.7.1.2, 11.3, 13.6.2). [Example:

   template<class T> struct A {
   struct C {
       template<class T2> struct B { };
       template<class T2> struct B<T2**> { }; // partial specialization #1
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]

A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

Within the argument list of a class template partial specialization, the following restrictions apply:

(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
```
— end example]

(9.2) The specialization shall be more specialized than the primary template (13.6.5.2).

(9.3) The template parameter list of a specialization shall not contain default template argument values.\(^{137}\)

(9.4) An argument shall not contain an unexpanded pack. If an argument is a pack expansion (13.6.3), it shall be the last argument in the template argument list.

The usual access checking rules do not apply to non-dependent names used to specify template arguments of the simple-template-id of the partial specialization. [Note: 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]

13.6.5.1 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.

(1.1) If exactly one matching specialization is found, the instantiation is generated from that specialization.

\(^{137}\) There is no way in which they could be used.
If more than one matching specialization is found, the partial order rules (13.6.5.2) 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.

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 (13.9.2), and the deduced template arguments satisfy the associated constraints of the partial specialization, if any (13.4.2). [Example:

```c
 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 T1, class T2, int I> 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*, I> 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:

```c
 template< typename T> concept C = requires (T t) { t.f(); };
 template< typename T> struct S { }; // #1
 template< C T> struct S<T> { }; // #2
```

```c
 struct Arg { void f(); }; // #1; the constraints of #2 are not satisfied
 struct S<int> s1; // uses #1; both constraints are satisfied but #2 is more specialized
 struct S<Arg> s2; // uses #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:

```c
 template <int I, int J> struct A {}; // error
 template <int I> struct A<int+5, I*2> {}; // OK
```

```c
 template <int I> struct A<int, I> {}; // OK
 template <int I, int J, int K> struct B {}; // OK
 template <int I> struct B<int, 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.

13.6.5.2 Partial ordering of class template specializations [temp.class.order]

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 (13.6.6.2):

(1.1) — Each of the two function templates has the same template parameters and associated constraints (13.4.2) as the corresponding partial specialization.

(1.2) — 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 { }; // OK
```
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:

```cpp
template<typename T> concept C = requires (T t) { t.f(); }

template<typename T> concept D = C<T> && requires (T t) { t.f(); }

template<typename T> class S { }

template<typename T> class S<T> { }; // #1

template<typename T> class S<T> { }; // #2

template<typename T> class S<T> { }; // #3

template<typename T> class S<T> { }; // #4

template<typename T> void f(S<T>); // A

template<typename T> void f(S<T>); // B

template<typename T> void g(Y<T>); // C

template<typename T> void g(Y<T>); // D
```

The partial specialization #2 is more specialized than #1 because B is more specialized than A. — end example]

### 13.6.5.3 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:

```cpp
// 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>::f() { }

// explicit specialization
template<> void A<char,2>::f() { }
```

```cpp
int main() {
    A<char,0> a0;
}```
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 (13.8.1,
13.8.2), 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]

13.6.6 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 (9.2.3.5).
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.\(^\text{138}\)

13.6.6.1 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:

```cpp
// translation unit 1: // translation unit 2:
template<class T>
void f(T*);
void g(int* p) {
    f(p); // calls f<int*>(int*)
}
```
—end example]

Such specializations are distinct functions and do not violate the one-definition rule (6.2).

The signature of a function template is defined in Clause 3. The names of the template parameters are
significant only for establishing the relationship between the template parameters and the rest of the signature.
[Note: Two distinct function templates may have identical function return types and function parameter lists,
even if overload resolution alone cannot distinguish them.

\(^{138}\) 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 (6.2) in a program, it must be defined; it will not be implicitly
instantiated using the function template definition.
template<class T> void f();

/* OK: overloads the first template */
template<int I> void f();
/* distinguishesw ith an explicit template argument list */

/* end note */

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. [Example:

```cpp
template <int I, int J> A<I+J> f(A<I>, A<J>); // #1
```  
```cpp
template <int K, int L> A<K+L> f(A<K>, A<L>); // same as #1
```  
```cpp
template <int I, int J> A<I-J> f(A<I>, A<J>); // different from #1
```  
/* end example */ [Note: Most expressions that use template parameters use non-type template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the `sizeof` operator. — end note]

Two expressions involving template parameters are considered equivalent if two function definitions containing the expressions would satisfy the one-definition rule (6.2), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. Two `lambda-expression`s are never considered equivalent. [Note: The intent is to avoid `lambda-expression`s appearing in the signature of a function template with external linkage. — end note] For determining whether two dependent names (13.7.2) are equivalent, only the name itself is considered, not the result of name lookup in the context of the template. If multiple declarations of the same function template differ in the result of this name lookup, the result for the first declaration is used. [Example:

```cpp
template <int I, int J> void f(A<I+J>);
```  
```cpp
template <int K, int L> void f(A<K+L>); // same as #1
```  
```cpp
int i = h<int>();
```  
```cpp
int g(int);
```  
```cpp
// template argument substitution fails; g(int)
```  
```cpp
} // was not in scope at the first declaration of h()
```  
```cpp
// ill-formed, no diagnostic required: the two expressions are functionally equivalent but not equivalent
```  
```cpp
template <int N> void foo(const char (*s)[sizeof(T)]);
```  
```cpp
// two different declarations because the non-dependent portions are not considered equivalent
```  
```cpp
// ill-formed, no diagnostic required: the two expressions are functionally equivalent but not equivalent
```  
```cpp
template <class T> void spam(decltype([N] {} (*s)[sizeof(T)]));
```  
```cpp
// ill-formed, no diagnostic required: the two expressions are functionally equivalent but not equivalent
```  
/* end example */ Two 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 `template-head`s are equivalent if their `template-parameter-list`s have the same length, corresponding `template-parameters` are equivalent, and if either has a `requires-clause`, they both have `requires-clauses` and the corresponding `constraint-expressions` are equivalent. Two `template-parameters` are equivalent under the following conditions:

1. they declare template parameters of the same kind,
2. if either declares a template parameter pack, they both do,
3. if they declare non-type template parameters, they have equivalent types,
4. if they declare template template parameters, their template parameters are equivalent, and
5. if either is declared with a `qualified-concept-name`, they both are, and the `qualified-concept-names` are equivalent.
When determining whether types or qualified-concept-names 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 (13.4.1) 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]

13.6.6.2 Partial ordering of function templates [temp.func.order]

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (13.9.2) 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:

(1.1) — during overload resolution for a call to a function template specialization (12.3.3);
(1.2) — when the address of a function template specialization is taken;
(1.3) — when a placement operator delete that is a function template specialization is selected to match a placement operator new (6.6.5.4.2, 7.6.2.5);
(1.4) — when a friend function declaration (13.6.4), an explicit instantiation (13.8.2) or an explicit specialization (13.8.3) refers to a function template specialization.

Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function type. The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process. If both deductions succeed, the partial ordering selects the more constrained template as described by the rules in 13.4.4.

To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (13.6.3) 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] If only one of the function templates M is a member function, and that function is a non-static member of some class A, M is considered to have a new first parameter inserted in its function parameter list. Given cv as the cv-qualifiers of M (if any), the new parameter is of type “rvalue reference to cv A” if the optional ref-qualifier of M is && or if M has no ref-qualifier and the first parameter of the other template has rvalue reference type. Otherwise, the new parameter is of type “value 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:

```cpp
struct A { };
```
template<class T> struct B {
    template<class R> int operator*(R&); // #1
};

template<class T, class R> int operator*(T&, R&); // #2

// The declaration of B::operator* is transformed into the equivalent of
// template<class R> int operator*(B<A>&, R&); // #1a

int main() {
    A a;
    B<A> b;
    b * a; // calls #1a
}

— end example]

4 Using the transformed function template’s function type, perform type deduction against the other template
as described in 13.9.2.4.

[Example:

    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() {
        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(T*)
        A<int> z;
        h(z); // overload resolution selects h(A<T>&)
        const A<int> z2;
        h(z2); // h(const T&) is called because h(A<T>&) is not callable
    }

— end example]

5 [Note: Since partial ordering in a call context 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 U> void f(U, A<U, U>&* p = 0); // #2
    template<class T> void g(T, T = T()); // #3

§ 13.6.6.2
template<class T, class... U> void g(T, U...); // #4

do 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]

13.6.7 Alias templates [temp.alias]

1 A template-declaration in which the declaration is an alias-declaration (Clause 9) 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.

2 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 type-id of the alias template. [Note: An alias template name is never deduced. — end note] [Example:

    template<class T> struct Alloc { /* ... */
    template<class T> using Vec = vector<T, Alloc<T>>;
    Vec<int> v; // same as vector<int, Alloc<int>> v;

    template<class T>
    void process(Vec<T>& v)
    { /* ... */

    template<class T>
    void process(vector<T, Alloc<T>>& w)
    { /* ... */ // error: redefinition

    template<template<class> class TT>
    class TT>
    void f(TT<int>);

    f(v); // error: Vec not deduced

    template<template<class,class> class TT>
    void g(TT<int, Alloc<int>>);
    g(v); // OK: TT = vector

    — end example]

3 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> using void_t<typename T::foo> f();
    f<int>(); // error, int does not have a nested type foo

    — end example]

4 The 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]

13.6.8 Concept definitions

A concept is a template that defines constraints on its template arguments.
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 } -> 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 (6.3.6).
A concept shall not have associated constraints (13.4.2).
A concept is not instantiated (13.8). [Note: An id-expression that denotes a concept specialization is evaluated as an expression (7.5.4). A concept cannot be explicitly instantiated (13.8.2), explicitly specialized (13.8.3), or partially specialized. —end note]

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.

13.7 Name resolution

Three kinds of names can be used within a template definition:

(1.1) — The name of the template itself, and names declared within the template itself.
(1.2) — Names dependent on a template-parameter (13.7.2).
(1.3) — 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:

    // 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
    }

    —end example]
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;
// T::A is not a type name:
// multiplication of T::A by a7; ill-formed, no visible declaration of a7
B* a8;
// B is not a type name:
// multiplication of B by a8; ill-formed, no visible declarations of B and a8

—end example

typename-specifier:
   typename nested-name-specifier identifier
   typename nested-name-specifier template
   simple-template-id

A typename-specifier denotes the type or class template denoted by the simple-type-specifier (9.1.7.2) formed by omitting the keyword typename. The usual qualified name lookup (6.4.3) is used to find the qualified-id even in the presence of typename. [Example:

struct A {
   struct X { }; // struct A::X
   int X;
};
struct B {
   struct X { }; // struct B::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 (11.6) 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

— it is a qualified name in a type-id-only context (see below), or
— it is a decl-specifier of the decl-specifier-seq of a
— simple-declaration or a function-definition in namespace scope,
— member-declaration,
— parameter-declaration in a member-declaration\(^\text{139}\), unless that parameter-declaration appears in a default argument,
— 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,
— parameter-declaration in a lambda-declarator, unless that parameter-declaration appears in a default argument, or
— 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:

\(^\text{139}\) This includes friend function declarations.
template<class T> T::R f();  // OK, return type of a function declaration at global scope
template<class T> void f(T::R);   // ill-formed (no diagnostic required), attempt to declare
// a void variable template

template<class T> struct S {
  using Ptr = PtrTraits<T>::Ptr;  // OK, in a defining-type-id
  T::R f(T::P p) {               // OK, class scope
    return static_cast<T::R>(p); // OK, type-id of a static_cast
  }
  auto g() -> S<T>::Ptr;        // OK, trailing-return-type
};

template<typename T> void f() { // variable pf of type void* initialized with T::X
  void (*pf)(T::X);
  // error: T::X at block scope does not denote a type
  void g(T::X);                  // (attempt to declare a void variable)
}

— 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:

```c
template <class T> void f(int i) {
  T::x * i; // expression, not the declaration of a variable i
}
```

```c
struct Foo {
  typedef int x;
};

struct Bar {
  static int const x = 5;
};

int main() {
  f<Bar>(1); // OK
  f<Foo>(1);   // error: Foo::x is a type
}
— 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 (13.7.2.1). [Example:

```c
template<class T> struct A {
  typedef int B;
  B b;   // OK, no typename required
};

— end example —
```

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
(8.4.1) 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:
— 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

— 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

— 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

— constant expression evaluation (7.7) within the template instantiation uses

— the value of a const object of integral or unscoped enumeration type or

— the value of a constexpr object or

— the value of a reference or

— the definition of a constexpr function,

and that entity was not defined when the template was defined, or

— 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]

9 When looking for the declaration of a name used in a template definition, the usual lookup rules (6.4.1, 6.4.2) are used for non-dependent names. The lookup of names dependent on the template parameters is postponed until the actual template argument is known (13.7.2). [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';
  }
};

§ 13.7

373
In the example, \( i \) is the local variable \( i \) declared in \texttt{printall}, \( \texttt{cnt} \) is the member \( \texttt{cnt} \) declared in \texttt{Set}, and \( \texttt{cout} \) is the standard output stream declared in \texttt{iostream}. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual \texttt{template-arguments} are known. For example, even though the name \texttt{operator\textless\textgreater} is known within the definition of \texttt{printall()} and a declaration of it can be found in \texttt{<iostream>}, the actual declaration of \texttt{operator\textless\textgreater} needed to print \texttt{p[i]} cannot be known until it is known what type \( T \) is (13.7.2). — end example"

10 If a name does not depend on a \texttt{template-parameter} (as defined in 13.7.2), 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(t); // dependent
  dd++;
  // not dependent; error: declaration for dd not found
}
```

— end example]

11 [Note: For purposes of name lookup, default arguments and \texttt{noexcept-specifiers} of function templates and default arguments and \texttt{noexcept-specifiers} of member functions of class templates are considered definitions (13.6). — end note]

13.7.1 Locally declared names

Like normal (non-template) classes, class templates have an injected-class-name (Clause 11). The injected-class-name can be used as a \texttt{template-name} or a \texttt{type-name}. When it is used with a \texttt{template-argument-list}, as a \texttt{template-argument} for a \texttt{template-parameter}, or as the final identifier in the \texttt{elaborated-type-specifier} of a friend class template declaration, it is a \texttt{template-name} that refers to the class template itself. Otherwise, it is a \texttt{type-name} equivalent to the \texttt{template-name} followed by the \texttt{template-parameters} of the class template enclosed in `\texttt{<>}`.

Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a \texttt{type-name}, it is equivalent to the \texttt{template-name} followed by the \texttt{template-arguments} of the class template specialization or partial specialization enclosed in `\texttt{<>}`. [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 Y> friend class Y; // meaning ::Y
  }
};
```

— end example]

The injected-class-name of a class template or class template specialization can be used as either a \texttt{template-name} or a \texttt{type-name} wherever it is in scope. [Example:

```cpp
template <class T> struct Base { 
  Base* p;
};
```
template <class T> struct Derived: public Base<T> {
    typename Derived::Base* p;       // meaning Derived::Base<T>
};

template<class T, template<class> class U = T::template Base> struct Third { }; Third<Derived<int> > t; // OK: default argument uses injected-class-name as a template

—end example

4 A lookup that finds an injected-class-name (11.7) 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
typein 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 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;
    void f() {               // error: template-parameter redeclared
        char T;
    }
};
template<class X> class X;             // error: template-parameter redeclared
—end example
```

7 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(); 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
}
§ 13.7.1 375
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<C>::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 (13.7.2.1), 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 (6.3.10). [Example:

```cpp
struct A {
    struct B { /* ... */ }
    int a;
    int Y;
};

template<class B, class a> struct X : A {
    B b;
    a b;       // error: A's a isn't a type name
};
```
—end example]

### 13.7.2 Dependent names [temp.dep]

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.

1 In an expression of the form:

\[
\text{postfix-expression ( expression-list_{opt} )}
\]

where the postfix-expression is an unqualified-id, the unqualified-id denotes a dependent name if

1. any of the expressions in the expression-list is a pack expansion (13.6.3),
2. any of the expressions or braced-init-lists in the expression-list is type-dependent (13.7.2.2), 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 (13.7.4.1) in both the context of the template definition and the context of the point of instantiation (13.7.4.2). — end note]

3 [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 (13.7.2.1) 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]  

```cpp
struct A {
    struct B { /* ... */ };  // a has type double
    int a;
    int Y;
};

int a;

template<class T> struct Y : T {
    struct 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]

13.7.2.1 Dependent types [temp.dep.type]

1 A name refers to the current instantiation if it is

1.1 — in the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, the injected-class-name (Clause 11) of the class template or nested class,

1.2 — in the definition of a primary class template or a member of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in <> (or an equivalent template alias specialization),

1.3 — in the definition of a nested class of a class template, the name of the nested class referenced as a member of the current instantiation, or

1.4 — in the definition of a partial specialization or a member of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <> (or an equivalent template alias specialization). If the n\text{th} template parameter is a template parameter pack, the n\text{th} template argument is a pack expansion (13.6.3) whose pattern is the name of the template parameter pack.

2 The template argument list of a primary template is a template argument list in which the n\text{th} template argument has the value of the n\text{th} template parameter of the class template. If the n\text{th} template parameter is a template parameter pack (13.6.3), the n\text{th} template argument is a pack expansion (13.6.3) whose pattern is the name of the template parameter pack.

3 A template argument that is equivalent to a template parameter can be used in place of that template parameter in a reference to the current instantiation. For a template type-parameter, a template argument
is equivalent to a template parameter if it denotes the same type. For a non-type template parameter, a template argument is equivalent to a template parameter if it is an identifier that names a variable that is equivalent to the template parameter. A variable is equivalent to a template parameter if

(3.1) — it has the same type as the template parameter (ignoring cv-qualification) and

(3.2) — 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:]

```cpp
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>::* p3; // A<T>::* is not the current instantiation
    };
};

template <class T> class A<T*> {
    A<T*>* p1; // A<T*> is the current instantiation
    A<T> p2; // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
    B<T1, T2, I>* b1; // refers to the current instantiation
    B<T2, T1, I>* b2; // not the current instantiation
    typedef T1 my_T1;
    static const int my_I = I;
    static const int my_I2 = I+0;
    static const int my_I3 = my_I;
    static const long my_I4 = I;
    static const int my_I5 = (I);
    B<my_T1, T2, my_I1> b3; // refers to the current instantiation
    B<my_T1, T2, my_I2> b4; // not the current instantiation
    B<my_T1, T2, my_I3> b5; // refers to the current instantiation
    B<my_T1, T2, my_I4> b6; // not the current instantiation
    B<my_T1, T2, my_I5> b7; // not the current instantiation
};
```

—end example]

4 A dependent base class is a base class that is a dependent type and is not the current instantiation. [Note: A base class can be the current instantiation in the case of a nested class naming an enclosing class as a base. [Example:]

```cpp
template<class T> struct A {
    typedef int M;
    struct B {
        typedef void M;
        struct C;
    };
};

template<class T> struct A<T>::B::C : A<T> {
    M m; // OK, A<T>::M
};
```

—end example] —end note]

5 A name is a member of the current instantiation if it is
(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 (6.4.5), 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.

6 A name is a member of an unknown specialization if it is

(6.1) A qualified-id in which the nested-name-specifier names a dependent type that is not the current instantiation.

(6.2) A qualified-id in which the nested-name-specifier refers to the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the qualified-id does not find any member of a class that is the current instantiation or a non-dependent base class thereof.

(6.3) An id-expression denoting the member in a class member access expression (7.6.1.4) in which either

(6.3.1) 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

(6.3.2) the type of the object expression is not the current instantiation and the object expression is type-dependent.

7 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:

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
    }
};
— end example]
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 11.3.2
```

—end example—

A type is dependent if it is

1. a template parameter,
2. a member of an unknown specialization,
3. a nested class or enumeration that is a dependent member of the current instantiation,
4. a cv-qualified type where the cv-unqualified type is dependent,
5. a compound type constructed from any dependent type,
6. an array type whose element type is dependent or whose bound (if any) is value-dependent,
7. a function type whose exception specification is value-dependent,
8. denoted by a simple-template-id in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent or is a pack expansion [Note: This includes an injected-class-name (Clause 11) of a class template used without a template-argument-list. — end note] , or
9. denoted by decltype(expression), where expression is type-dependent (13.7.2.2).

[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]

13.7.2.2 Type-dependent expressions

Except as described below, an expression is type-dependent if any subexpression is type-dependent.

1. this is type-dependent if the class type of the enclosing member function is dependent (13.7.2.1).
2. An id-expression is type-dependent if it contains
   1. an identifier associated by name lookup with one or more declarations declared with a dependent type,
   2. an identifier associated by name lookup with a non-type template-parameter declared with a type that contains a placeholder type (9.1.7.5),
   3. an identifier associated by name lookup with a variable declared with a type that contains a placeholder type (9.1.7.5) where the initializer is type-dependent,
   4. an identifier associated by name lookup with one or more declarations of member functions of the current instantiation declared with a return type that contains a placeholder type,
   5. an identifier associated by name lookup with a structured binding declaration (9.5) whose brace-or-equal-initializer is type-dependent,
— the identifier \texttt{\_\_func\_} (9.4.1), where any enclosing function is a template, a member of a class template, or a generic lambda,

— the identifier introduced in a postcondition (9.11.4) to represent the result of a templated function whose declared return type contains a placeholder type,

— a \texttt{template-id} that is dependent,

— a \texttt{conversion-function-id} that specifies a dependent type, or

— a \texttt{nested-name-specifier} or a \texttt{qualified-id} that names a member of an unknown specialization;

or if it names a dependent member of the current instantiation that is a static data member of type “array of unknown bound of \texttt{T}” for some \texttt{T} (13.6.1.3). 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-initializer_opt
::opt new new-placement_opt ( type-id ) new-initializer_opt
dynamic_cast < type-id > ( expression )
static_cast < type-id > ( expression )
const_cast < type-id > ( expression )
reinterpret_cast < type-id > ( expression )
( type-id ) cast-expression
\end{verbatim}

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
::opt delete [ ] cast-expression
throw assignment-expression_opt
noexcept ( expression )
\end{verbatim}

[\textit{Note}: For the standard library macro \texttt{offsetof}, see 17.2. — end note]

5 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. [\textit{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]

6 A \texttt{braced-init-list} is type-dependent if any element is type-dependent or is a pack expansion.

7 A \texttt{fold-expression} is type-dependent.

\section*{13.7.2.3 Value-dependent expressions} [temp.dep.constexpr]

Except as described below, an expression used in a context where a constant expression is required is value-dependent if any subexpression is value-dependent.

An \texttt{id-expression} is value-dependent if:

\begin{enumerate}
\item it is type-dependent,
\item it is the name of a non-type template parameter,
\item it names a static data member that is a dependent member of the current instantiation and is not initialized in a \texttt{member-declarator},
\item it names a static member function that is a dependent member of the current instantiation, or
\item it is a constant with literal type and is initialized with an expression that is value-dependent.
\end{enumerate}
Expressions of the following form are value-dependent if the unary-expression or expression is type-dependent or the type-id is dependent:

- `sizeof unary-expression`
- `sizeof ( type-id )`
- `typeid ( expression )`
- `typeid ( type-id )`
- `alignof ( type-id )`
- `noexcept ( expression )`

[Note: For the standard library macro `offsetof`, see 17.2. — end note]

Expressions of the following form are value-dependent if either the type-id or simple-type-specifier is dependent or the expression or cast-expression is value-dependent:

- `simple-type-specifier ( expression-list_opt )`
- `static_cast < type-id > ( expression )`
- `const_cast < type-id > ( expression )`
- `reinterpret_cast < type-id > ( expression )`
- `( type-id ) cast-expression`

Expressions of the following form are value-dependent:

- `sizeof ... ( identifier )`
- `fold-expression`

An expression of the form `&qualified-id` where the qualified-id names a dependent member of the current instantiation is value-dependent. An expression of the form `&cast-expression` is also value-dependent if evaluating `cast-expression` as a core constant expression (7.7) succeeds and the result of the evaluation refers to a templated entity that is an object with static or thread storage duration or a member function.

13.7.2.4 Dependent template arguments

1 A type template-argument is dependent if the type it specifies is dependent.

2 A non-type template-argument is dependent if its type is dependent or the constant expression it specifies is value-dependent.

3 Furthermore, a non-type template-argument is dependent if the corresponding non-type template-parameter is of reference or pointer type and the template-argument designates or points to a member of the current instantiation or a member of a dependent type.

4 A template template-argument is dependent if it names a template-parameter or is a qualified-id that refers to a member of an unknown specialization.

13.7.3 Non-dependent names

1 Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [Example:

```cpp
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]

13.7.4 Dependent name resolution

13.7.4.1 Point of instantiation

1 For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated
because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

2 If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

3 For a noexcept-specifier of a function template specialization or specialization of a member function of a class template, if the noexcept-specifier is implicitly instantiated because it is needed by another template specialization and the context that requires it depends on a template parameter, the point of instantiation of the noexcept-specifier is the point of instantiation of the specialization that requires it. Otherwise, the point of instantiation for such a noexcept-specifier immediately follows the namespace scope declaration or definition that requires the noexcept-specifier.

4 For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

5 If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

6 An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

7 A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit, and in addition to the points of instantiation described above,

   (7.1) — for any such specialization that has a point of instantiation within the top-level-declaration-seq of the translation unit, prior to the private-module-fragment (if any), the point after the top-level-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 (6.2), the program is ill-formed, no diagnostic required.

13.7.4.2 Candidate functions

1 For a function call where the postfix-expression is a dependent name, the candidate functions are found using the usual lookup rules from the template definition context (6.4.1, 6.4.2). [Note: For the part of the lookup using associated namespaces (6.4.2), function declarations found in the template instantiation context are found by this lookup, as described in 6.4.2. — 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.

2 [Example:

Source file "X.h":

```c
namespace Q {
    struct X { };
}
```

Source file "G.h":

```c
namespace Q {
    void g_impl(X, X);
}
```

§ 13.7.4.2 383
Module interface unit of M1:

```c
module;
#include "X.h"
#include "G.h"
export module M1;
export template<typename T>
void g(T t) {
    g_impl(t, Q::X{ }); // ADL in definition context finds Q::g_impl, g_impl not discarded
}
```

Module interface unit of M2:

```c
module;
#include "X.h"
export module M2;
import M1;
void h(Q::X x) {
    g(x);
    // OK
}
```

— end example

Example:

Module interface unit of Std:

```c
export module Std;
export template<typename Iter>
void indirect_swap(Iter lhs, Iter rhs)
{
    swap(*lhs, *rhs); // swap not found by unqualified lookup, can be found only via ADL
}
```

Module interface unit of M:

```c
export module M;
import Std;
struct S { /* ... */};
void swap(S& p, S& q) {
    indirect_swap(p, q); // finds #1 via ADL in instantiation context
}
```

— end example

Example:

Source file "X.h":

```c
struct X { /* ... */};
X operator+(X, X);
```

Module interface unit of F:

```c
export module F;
export template<typename T>
void f(T t) {
    t + t;
}
```

Module interface unit of M:

```c
module;
#include "X.h"
export module M;
import F;
void g(X x) {
    f(x); // OK: instantiates t from F,
}
// operator+ is visible in instantiation context

— end example] 5

[Example:

Module interface unit of A:

```c
export module A;
export template<typename T>
void f(T t) {
    cat(t, t);  // #1
    dog(t, t);  // #2
}
```

Module interface unit of B:

```c
export module B;
import A;
export template<typename T, typename U>
void g(T t, U u) {
    f(t);
}
```

Source file "foo.h", not an importable header:

```c
struct foo {
    friend int cat(foo, foo);
};
int dog(foo, foo);
```

Module interface unit of C1:

```c
module;
#include "foo.h"  // dog not referenced, discarded
export module C1;
import B;
export template<typename T>
void h(T t) {
    g(foo{ }, t);
}
```

Translation unit:

```c
import C1;
void i() {
    h(0);  // error: dog not found at #2
}
```

Importable header "bar.h":}

```c
struct bar {
    friend int cat(bar, bar);
};
int dog(bar, bar);
```

Module interface unit of C2:

```c
module;
#include "bar.h"  // imports header unit "bar.h"
export module C2;
import B;
export template<typename T>
void j(T t) {
    g(bar{ }, t);
}
```

Translation unit:

```c
import C2;
void k() {
    j(0);  // OK, dog found in instantiation context:
}
```

§ 13.7.4.2
13.7.5 Friend names declared within a class template

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 (11.8.3). Such names may be found under the rules for associated classes (6.4.2).

Example:

```c++
template<typename T> struct number {
  number(int);
  friend number gcd(number x, number y) { return 0; };
};

void g() {
  number<double> a(3), b(4);
  a = gcd(a,b); // finds gcd because number<double> is an associated class,
  // making gcd visible in its namespace (global scope)
  b = gcd(3,4); // ill-formed; gcd is not visible
}
```

13.8 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`. Example:

```c++
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<int> { }; // specialize for T == int
template<> void g(char) { } // specialize for U == char
  // U is deduced from the parameter type
template<> void g<int>(int) { } // specialize for U == int
template<> int A<char>::x = 0; // specialize for T == char
```

140) Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.
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 (13.8.1) for a given argument list or be explicitly instantiated (13.8.2). A specialization is a class, variable, function, or class member that is either instantiated or explicitly specialized (13.8.3).

For a given template and a given set of template-arguments,

(5.1) — an explicit instantiation definition shall appear at most once in a program,
(5.2) — an explicit specialization shall be defined at most once in a program, as specified in 6.2, and
(5.3) — both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

An implementation is not required to diagnose a violation of this rule.

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:

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 (13.7.2.1) 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; // ill-formed: would declare A<function>::t as a static member function

— end example]

13.8.1 Implicit instantiation

Unless a class template specialization has been explicitly instantiated (13.8.2) or explicitly specialized (13.8.3), 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

— end example [If a class template has been declared, but not defined, at the point of instantiation (13.7.4.1),
the instantiation yields an incomplete class type (6.7).] [Example:

    template<class T> class X;
    X<char> ch; // error: incomplete type

— end example] [Note: Within a template declaration, a local class (11.5) 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]

2 The implicit instantiation of a class template specialization causes

(2.1) — 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

(2.2) — 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:

    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 6.2 and 11.3, a declaration that corresponds to a definition in the template is considered to be a
definition. [Example:

    template<class T, class U>
    struct Outer {
        template<class X, class Y> struct Inner;
        template<class Y> struct Inner<T, Y> { }; // #1a
        template<class Y> struct Inner<T, Y> { }; // #1b; OK: valid redeclaration of #1a
        template<class Y> struct Inner<Y, Y> { }; // #2
    };
    Outer<int, int> outer; // error at #2

Outer<int, int>::Inner<int, Y> is redeclared at #1b. (It is not defined but noted as being associated
with a definition in Outer<T, U>.) #2 is also a redeclaration of #1a. It is noted as associated with a
definition, so it is an invalid redeclaration of the same partial specialization.

    template<typename T> struct Friendly {
        template<typename U> friend int f(U) { return sizeof(T); }
    };
    Friendly<char> fc;
    Friendly<float> ff; // ill-formed: produces second definition of f(U)

— end example]

3 Unless a member of a class template or a member template has been explicitly instantiated or explicitly
specialized, 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.

4 Unless a function template specialization has been explicitly instantiated or explicitly specialized, 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.

5 [Example:]

```
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]

6 Unless a variable template specialization has been explicitly instantiated or explicitly specialized, 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.

7 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:]

```
template<typename T> constexpr int f() { return T::value; }
template<bool B, typename T> void g(decltype(B ? f<T>() : 0));
template<bool B, typename T> void g(...);
template<bool B, typename T> void h(decltype(int{B ? f<T>() : 0}));
template<bool B, typename T> void h(...);

void x() {
    g<false, int>(0); // OK, B ? f<T>() : 0 is not potentially constant evaluated
    h<false, int>(0); // error, instantiates f<int> even though B evaluates to false and
    // list-initialization of int from int cannot be narrowing
}
```

—end example]

8 If the function selected by overload resolution (12.3) can be determined without instantiating a class template definition, it is unspecified whether that instantiation actually takes place. [Example:]

```
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]

9 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 (13.9.3).

10 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 (8.4.1), 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 instantiations in the default argument to be implicitly instantiated.

11 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:

    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]

12 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 \( f \).

13 Each default argument is instantiated independently. [Example:

    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
The `noexcept-specifier` of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (14.4). If such an `noexcept-specifier` is needed but has not yet been instantiated, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the `noexcept-specifier` is done as if it were being done as part of instantiating the declaration of the specialization at that point.

[Note: 13.7.4.1 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-constraints` and `requires-clause` of a template specialization or member function are not instantiated along with the specialization or function itself, even for a member function of a local class; substitution into the atomic constraints formed from them is instead performed as specified in 13.4.2 and 13.4.1.2 when determining whether the constraints are satisfied. [Note: The satisfaction of constraints is determined during name lookup or overload resolution (12.3). — 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
};
```

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>` (11.3.4.1), 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
};
```

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]

### 13.8.2 Explicit instantiation

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. An explicit instantiation of a function template,
member function of a class template, or variable template shall not use the inline, constexpr, or constexpr
specifiers.

2 The syntax for explicit instantiation is:

```plaintext
explicit-instantiation:
    extern_opt template declaration
```

There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation
declaration. An explicit instantiation declaration begins with the extern keyword.

3 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
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 (9.7.1), any namespace from its enclosing namespace set. [Note: Regarding qualified
names in declarators, see 9.2.3. — end note] [Example:

```plaintext
template<class T> class Array { void mf(); }
template class Array<char>;
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);    // argument is deduced here

namespace N {
    template<class T> void f(T&) { }
}
template void N::f<int>(int&);
} — end example]
```

4 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 (11.3.3), 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 6.5, except as specified in 14.4. [Example:

```plaintext
template<typename T> T var = {};  // OK, instantiated variable has type float
template int float var<float>;    // OK, absence of major array bound is permitted
template int var<int[16]>[];      // error: instantiated variable has type int

template<typename T> auto av = T();
template int av<int>;

template<typename T> auto f() {}  // error: function with deduced return type
template void f<int>();           // redeclared with non-deduced return type (9.1.7.5)
} — 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.

§ 13.8.2 392
For a given set of template arguments, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, for an explicit instantiation definition the definition of a function template, a variable template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

An explicit instantiation of a class, function template, or variable template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [Example:

```cpp
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 (13.9.2). [Example:

```cpp
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 shall satisfy that template’s associated constraints (13.4.2). The satisfaction of constraints is determined when forming the template name of an explicit instantiation in which all template arguments are specified (13.2), or, for explicit instantiations of function templates, during template argument deduction (13.9.2.6) 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 (13.4.2, 13.4.1), 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.

Except for inline functions and variables, declarations with types deduced from their initializer or return value (9.1.7.5), const variables of literal types, variables of reference types, and class template specializations, explicit instantiation declarations have the effect of suppressing the implicit instantiation of the definition of the entity to which they refer. [Note: The intent is that an inline function that is the subject of an explicit instantiation declaration will still be implicitly instantiated when odr-used (6.2) so that the body can be considered for inlining, but that no out-of-line copy of the inline function would be generated in the translation unit. — end note]

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 (13.8.1) in the translation unit shall be the subject of an explicit instantiation definition

§ 13.8.2
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.

14 An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is
not done. [Example:

    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]

13.8.3 Explicit specialization [temp.expl.spec]

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:

    explicit-specialization:
    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 chars; 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 may be declared in any scope in which the corresponding primary template may be
defined (9.7.1.2, 11.3, 13.6.2).

3 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]

§ 13.8.3 394
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 preclude 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 (11.3.3),
the program is ill-formed.

A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the
class template; instead, the member of the class template specialization shall itself be explicitly defined if its
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:

```cpp
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<> also not used when defining a member of an explicitly specialized member class
void A<char>::B::f() { /* ... */ }

// template<> is used when defining a member of an explicitly specialized member class template
// specialized as a class template
template<class U> void A<char>::C<U>::f() { /* ... */ }

// template<> not permitted
template<> template<class U> struct A<short>::C { void f();

// error: template<> required
template<class U> void A<short>::C<U>::f() { /* ... */ }
```
—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

§ 13.8.3
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:

```cpp
class String { }
template<class T> class Array { /* ... */ }
template<class T> void sort(Array<T>& v) { /* ... */ }

void f(Array<String>& v) {
    sort(v);  // use primary template
    sort(Array<T>&)  // T is String
}
```]

```cpp
template<> void sort<String>(Array<String>& v);  // error: specialization after use of primary template
template<> void sort<>(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
```

---end example---

7 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 class templates of non-template classes, static data member templates of non-template classes, member class templates 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.

8 A template explicit specialization is in the scope of the namespace in which the template was defined. [Example:

```cpp
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
}
```

---end example---

9 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 (6.7). [Example:

```cpp
template<class T> class X;  // X is a class template
template<class T> class X<int>;
X<int> * p;
X<int> x;  // error: object of incomplete class X<int>
```

---end example---

§ 13.8.3 396
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:

```cpp
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>&);
```

—end example]

[Note: An explicit specialization of a constrained template shall satisfy that template’s associated constraints (13.4.2). The satisfaction of constraints is determined when forming the template name of an explicit specialization in which all template arguments are specified (13.2), or, for explicit specializations of function templates, during template argument deduction (13.9.2.6) when one or more trailing template arguments are left unspecified. —end note]

A function with the same name as a template and a type that exactly matches that of a template specialization is not an explicit specialization (13.6.6).

An explicit specialization of a function or variable template is inline only if it is declared with the `inline` specifier or defined as deleted, and independently of whether its function or variable template is inline. [Example:

```cpp
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]

[Note: For an explicit specialization of a function template, the contract conditions (9.11.4) of the explicit specialization are independent of those of the primary template. —end note]

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`:

```cpp
template<> X Q<int>::x; // declaration
template<> X Q<int>::x () ; // error: declares a function
template<> X Q<int>::x { } ; // definition
```

—end note]

A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization. [Example:

```cpp
template<class T> struct A {
    void f(T);
    template<class X1> void g1(T, X1);
    template<class X2> void g2(T, X2);
    void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X1> void A<T>::g1(T, X1) { }

// member template specialization
template<> template<class X1> void A<int>::g1(int, X1);
```
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:

```cpp
template<class T1> class A {
    template<class T2> class B {
        void mf();
    };
};
template<> template<> class A<int>::B<double>;
template<> template<> void A<char>::B<char>::mf();

— end example]
```

In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template and some of its enclosing class templates may remain unspecialized, except that the declaration shall not explicitly specialize a class member template if its enclosing class templates are not explicitly specialized as well. In such 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 { 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() { } // ill-formed; 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]
13.9 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 13.9.2), 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>(int*)` has a static variable `s` of type `int` and `f<char*>(char**)` has a static variable `s` of type `char*`. — end example]

13.9.1 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.

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

template<class U, class V> U convert(V v);`n
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 (11.3.4, 6.4.3.1).

A template argument list may be specified when referring to a specialization of a function template

(3.1) — when a function is called,
(3.2) — when the address of a function is taken, when a function initializes a reference to function, or when a pointer to member function is formed,
(3.3) — in an explicit specialization,
(3.4) — in an explicit instantiation, or
(3.5) — in a friend declaration.

Trailing template arguments that can be deduced (13.9.2) or obtained from default template-arguments may be omitted from the list of explicit template-arguments. A trailing template parameter pack (13.6.3) 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);`n
```
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);  // ill-formed: X cannot be deduced
    f<void>(f<int, bool>);  // Y for outer f deduced as int (*)(bool)
    f<void>(f<int>);  // ill-formed: 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 (*)(bool), Z deduced as an empty sequence
}

—end example

5 [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 (9.2.3.5) is visible that would otherwise be used. For example:

```
template <class T> int f(T);  // #1
int f(int);  // #2
int k = f<int>(1);  // uses #2
int l = f<>(1);  // uses #1
```
—end note]

6 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:

```
template<class X, class Y, class Z> X f(Y,Z);
template<class ... Args> void f2();
void g() {
    f<int, const char*, double>("aa", 3.0);  // Z deduced as double
    f<int>("aa", 3.0);  // Y deduced as const char*; Z deduced as double
    f("aa", 3.0);  // error: X cannot be deduced
    f2<char, short, int, long>();  // OK
}
```
—end example]

7 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,

```
template<class T> void f(T);

class Complex {
    Complex(double);
};

void g() {
    f<Complex>(1);  // OK, means f<Complex>(Complex(1))
}
—end note]

8 [Note: Because the explicit template argument list follows the function template name, and because constructor templates (11.3.4) are named without using a function name (6.4.3.1), there is no way to provide an explicit template argument list for these function templates. — end note]

9 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:

```
template<class ... Types> void f(Types ... values);

void g() {
    f<int*, float*>(0, 0, 0);  // Types deduced as the sequence int*, float*, int
}
```

§ 13.9.1
13.9.2 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)
}
```

—end example]}

When an explicit template argument list is specified, if the given template-id is not valid (13.2), 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 9.2.3.5 are performed. [Example: A parameter type of “void (const int, int[5])” becomes “void(*)(int,int*)”. —end example] [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 note] [Example:

```cpp
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]

[Note: f<int>(1) and f<const int>(1) call distinct functions even though both of the functions called have the same function type. —end note]

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, class U = double>
void f(T t = 0, U u = 0);
```

```cpp
void g() {
    f(1, 'c');  // f<int,char>(1,'c')
    f(1);       // f<int,double>(1,0)
}
```
§ 13.9.2 402

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 (13.4.2), those constraints are checked for satisfaction (13.4.1). 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] [Example:

```cpp
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> typename T::X g(typename A<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 h(...) { }
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 example]

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]

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:

```cpp
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);
  void 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
  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>*);

§ 13.9.2 403
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.  

(11.6) — Attempting to create a reference to void.  

(11.7) — Attempting to create “pointer to member of T” when T is not a class type. [Example:  
    template <class T> int f(int T::*);  
    int i = f<int>(0);  
    — end example]  

(11.8) — Attempting to give an invalid type to a non-type template parameter. [Example:  
    template <class T, T*> struct S {};  
    template <class T> int f(S<T, T()>::*);  
    struct X {};  
    int i0 = f<X>(0);  
    — end example]  

(11.9) — Attempting to perform an invalid conversion in either a template argument expression, or an expression  
used in the function declaration. [Example:  
    template <class T, T*> int f(int);  
    int i2 = f<int,1>(0);  // can’t conv 1 to int*  
    — end example]  

(11.10) — 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]  

[Example: In the following example, assuming a signed char cannot represent the value 1000, a narrowing  
conversion (9.3.4) would be required to convert the template-argument of type int to signed char, therefore  
substitution fails for the second template (13.3.2).  

    template <int> int f(int);  
    template <signed char> int f(int);  
    int i1 = f<1000>(0);  // OK  
    int i2 = f<1>(0);  // ambiguous; not narrowing  

— end example]  

13.9.2.1 Deducing template arguments from a function call  

Template argument deduction is done by comparing each function template parameter type (call it P)  
that contains template-parameters that participate in template argument deduction with the type of the  
corresponding argument of the call (call it A) as described below. If removing references and cv-qualifiers from  
P gives std::initializer_list<P*> or P[N] for some P’ and N and the argument is a non-empty initializer  
list (9.3.4), then deduction is performed instead for each element of the initializer list independently, taking  
P’ as separate function template parameter types P’ and the ith initializer element as the corresponding  
argument. In the P[N] case, if N is a non-type template parameter, N is deduced from the length of the  

§ 13.9.2.1 404
initializer list. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (13.9.2.5). [Example:

```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
k({{1},{2},{3}});  // OK, N deduced as 3

template<int M, int N> void m(int const(&)[M][N]);  
m({{{1},{2},{3}}},Aggr());  // 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
```

— 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 (13.9.2.5), the type of that pack is never deduced. [Example:

```cpp
template<class ... Types> void f(Types& ...);

template<class T1, class ... Types> void g(T1, Types ...);

template<class T1, class ... Types> void g1(Types ..., T1);

void h(int x, float& y) {
  const int z = x;
  f(x, y, z);  // Types deduced as int, float, const int
  g(x, y, z);  // T1 deduced as int; Types deduced as float, int
  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:
template<class T> int f(const T&);
int n1 = f(5);  // calls f<int>(const int&)
const int i = 0;
int n2 = f(i);   // calls f<int>(const int&)
template <class T> int g(volatile T&);
int n3 = g(i);   // calls g<const int&>(const volatile int&)

— end 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 (12.3.1.8)). 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
```

```cpp
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.
```

```cpp
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] 4 In general, the deduction process attempts to find template argument values that will make the deduced A identical to A (after the type A is transformed as described above). However, there are three cases that allow a difference:

(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:

```cpp
template <typename... T> struct X;
template <> struct X<> {};  
template <typename T, typename... Ts>
   struct X<T, Ts...> : X<Ts...> {};  
struct D : X<int> {};  

template <typename... T>
int f(const X<T...>&);
int x = f(D());  // calls f<int>, not f<>
                 // B is X<>, C is X<int>
```

— end example] 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
When \( P \) is a function type, function pointer type, or pointer-to-member-function type:

- If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.

- If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for the deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a non-deduced context.

Example:

```cpp
// Only one function of an overload set matches the call so the function parameter is a deduced context.
template <class T> int f(T (*p)(T));
int g(int);
int i = f(g);  // calls f(int (*)(int))
```

Example:

```cpp
// Ambiguous deduction causes the second function parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g);  // calls f(int, int (*)(int))
```

Example:

```cpp
// The overload set contains a template, causing the second function parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);  // calls f(int, int (*)(int))
```

If deduction succeeds for all parameters that contain \textit{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.

Example:

```cpp
template <class T> struct Z {
    typedef typename T::x xx;
};
template <class T> typename Z<T>::xx f(void *, T);  // #1
template <class T> void f(int, T);  // #2
struct A {} a;
int main()
    f(1, a);  // OK, deduction fails for #1 because there is no conversion from int to void*
```

13.9.2.2 Deducing template arguments taking the address of a function template

Template arguments can be deduced from the type specified when taking the address of an overloaded function (12.4). The function template’s function type and the specified type are used as the types of \( P \) and \( A \), and the deduction is done as described in 13.9.2.5.
A placeholder type (9.1.7.5) 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.

13.9.2.3 Deducing conversion function template arguments

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 9.3, 12.3.1.5, and 12.3.1.6 for the determination of that type) as described in 13.9.2.5.

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.

If \( A \) is not a reference type:

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,
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. 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.

13.9.2.4 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 13.6.6.2. — 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.\(^{141}\)
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 (13.6.6.2) the function template’s function type is used.

Each type nominated above from the parameter template and the corresponding type from the argument template are used as the types of \( P \) and \( A \).

\(^{141}\) Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.
Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

(5.1) If \( P \) is a reference type, \( P \) is replaced by the type referred to.

(5.2) 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:

(7.1) If \( P \) is a cv-qualified type, \( P \) is replaced by the cv-unqualified version of \( P \).

(7.2) 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 13.9.2.5. 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 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:

```cpp
template<class... Args> void f(Args... args);  // #1
template<class T1, class... Args> void f(T1 a1, Args... args);  // #2
template<class T1, class T2> void f(T1 a1, T2 a2);  // #3
```

f();  // calls #1
f(1, 2, 3);  // calls #2
f(1, 2);  // calls #3; non-variadic template #3 is more specialized
          // than the variadic templates #1 and #2

—end example]

If, for a given type, the types are identical after the transformations above and both \( P \) and \( A \) were reference types (before being replaced with the type referred to above):

(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.

Function template \( F \) is at least as specialized as function template \( G \) if, for each pair of types used to determine the ordering, the type from \( F \) is at least as specialized as the type from \( G \). \( F \) is more specialized than \( G \) if \( F \) is at least as specialized as \( G \) and \( G \) is not at least as specialized as \( F \).

If, after considering the above, function template \( F \) is at least as specialized as function template \( G \) and vice-versa, and if \( G \) has a trailing function parameter pack for which \( F \) does not have a corresponding parameter, and if \( F \) does not have a trailing function parameter pack, then \( F \) is more specialized than \( G \).

In most cases, deduction fails if not all template parameters have values, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [Note: A template parameter used in a non-deduced context is considered used. —end note] [Example:

```cpp
template <class T> T f(int);  // #1
template <class T, class U> T f(U);  // #2
void g() {
  f<int>(1);  // calls #1
}
```

—end example]

[Note: 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:

```cpp
template<class ...> struct Tuple {  
};
```
Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it P) is compared with an actual type (call it A), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make P, after substitution of the deduced values (call it the deduced A), compatible with A.

In some cases, the deduction is done using a single set of types P and A, in other cases, there will be a set of corresponding types P and A. Type deduction is done independently for each P/A pair, and the deduced template argument values are then combined. If type deduction cannot be done for any P/A pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails. The type of a type parameter is only deduced from an array bound if it is not otherwise deduced.

A given type P can be composed from a number of other types, templates, and non-type values:

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<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 13.9.2.1, if P contains no template-parameters that appear in deduced contexts, no deduction is done, so P and A need not have the same form. — end note]

The non-deduced contexts are:

1. The nested-name-specifier of a type that was specified using a qualified-id.
2. The expression of a decldtype-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 argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (12.4), and one or more of the following apply:
   1. more than one function matches the function parameter type (resulting in an ambiguous deduction), or
   2. no function matches the function parameter type, or
   3. the set of functions supplied as an argument contains one or more function templates.
6. A function parameter for which the associated argument is an initializer list (9.3.4) but the parameter does not have a type for which deduction from an initializer list is specified (13.9.2.1). [Example:
template<class T> void g(T);
g({1,2,3});  // error: no argument deduced for T

— end example

(5.7)

A function parameter pack that does not occur at the end of the parameter-declaration-list.

6 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<T>::B<T2>, both T and T2 are non-deduced. Likewise, if a type is specified as A<1+J>::X<T>, I, J, and T are non-deduced. If a type is specified as void f(typename A<T>::B, A<T>), the T in A<T>::B is non-deduced but the T in A<T> is deduced. — end example]

7 [Example: Here is an example in which different 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:

```cpp
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:

```cpp
template<class T> void f(const T*) { }

int* p;
void s() {
    f(p);  // f(const int*)
}
```

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

```cpp
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:
where \( (T) \) represents a parameter-type-list (9.2.3.5) 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 \( T \), \( <i> \) represents template argument lists where at least one argument contains an \( i \) and \( <> \) represents template argument lists where no argument contains a \( T \) or an \( i \).

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 (13.9.2.4), if \( A_i \) was originally a pack expansion:

- if \( P \) does not contain a template argument corresponding to \( A_i \), then \( A_i \) is ignored;
- otherwise, if \( P_i \) is not a pack expansion, template argument deduction fails.

[Example:

```cpp
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, T2> { }; // #3
S<int, const int&> s; // both #2 and #3 match; #3 is more specialized
```

```cpp
template<class T, class... U> struct A { }; // #1
template<class T1, class T2, class... U> struct A<T1, T2*, U...> { }; // #2
template<class T1, class T2> struct A<T1, T2> { }; // #3
template struct A<int, int*>; // selects #2
```

—end example]

Similarly, if \( P \) has a form that contains \( (T) \), then each parameter type \( P_i \) of the respective parameter-type-list (9.2.3.5) 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 (13.9.2.2) or when deducing template arguments from a function declaration (13.9.2.6) and \( A_i \) is an lvalue reference, in which case the type of \( A_i \) is adjusted 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:
template <class T> void f(T&&);
template <> void f(int&) { } // #1
template <> void f(int&&) { } // #2

void g(int i) {
    f(i); // calls f<int&>(int&), i.e., #1
    f(0); // calls f<int>(int&&), i.e., #2
}

— end example]

If the parameter-declaration corresponding to \( P_i \) is a function parameter pack, then the type of its declarator-id is compared with each remaining parameter type in the parameter-type-list of \( A \). Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. During partial ordering (13.9.2.4), 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 \( R \) is 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] 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]

14 [Example:

```
template<class T, T i> void f(int (&a)[i]);
```

int v[10];
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:

```cpp
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);
template<int i> void f3(int (&a)[i][20]);
```

void g() {
    int v[10][20];
    f1(v);     // OK: i deduced as 20
    f1<20>(v);  // OK
    f2(v);     // error: cannot deduce template-argument i
    f2<10>(v);  // OK
    f3(v);     // OK: i deduced as 10
}

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 { /* ... */};
template <int i> void g(A<i+1>);  // i is not 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]

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]

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.142 [Example:

```cpp
template<int i> class A { /* ... */};
template<short s> void f(A<s>);
void k1() {
    A<1> a;
    f(a);       // error: deduction fails for conversion from int to short
```

142) 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.

§ 13.9.2.5
f<1>(a);                  // OK
}

template<const short cs> class B { }
template<short s> void g(B<s>);
void k2() {
  B<1> b;
  g(b);                  // OK: cv-qualifiers are ignored on template parameter types
}
— end example]

19 A template-parameter can be deduced from a function, pointer to function, or pointer-to-member-function type. [Example:

template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
void g(char,int);
void h(int,int,int);
void h(char,int);
int m() {
  f(&g);                  // error: ambiguous
  f(&h);                  // OK: void h(char,int) is a unique match
  f(&foo);                // error: type deduction fails because foo is a template
}
— end example]

20 A template-parameter cannot be deduced from the type of a function default argument. [Example:

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]

21 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:

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]

22 [Note: Template argument deduction involving parameter packs (13.6.3) can deduce zero or more arguments for each parameter pack. — end note] [Example:

template<class> struct X {  
};
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)>{  
};
template<class T, class ... Types> struct Y {  
};
template<class T, class ... Types> struct Y<T, Types& ...> {  
};

template<class ... Types> int f(void (*)(Types ...));
void g(int, float);

X<int> x1;                     // uses primary template
X<int(int, float, double)> x2;  // uses partial specialization; ArgTypes contains float, double
X<int(float, int)> x3;          // uses primary template
Y<int> y1;                     // use primary template; Types is empty
Y<int&, float&, double&> y2;    // uses partial specialization; T is int&, Types contains float, double
Y<int, float, double> y3;       // uses primary template; Types contains int, float, double

§ 13.9.2.5
int fv = f(g); // OK; Types contains int, float
—end example

13.9.2.6 Deducing template arguments from a function declaration [temp.deduct.decl]

In a declaration whose declarator-id refers to a specialization of a function template, template argument deduction is performed to identify the specialization to which the declaration refers. Specifically, this is done for explicit instantiations (13.8.2), explicit specializations (13.8.3), and certain friend declarations (13.6.4). This is also done to determine whether a deallocation function template specialization matches a placement operator new (6.6.5.4.2, 7.6.2.5). 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 7.6.2.5. The deduction is done as described in 13.9.2.5.

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 (13.6.6.2), deduction fails and, in the declaration cases, the program is ill-formed.

13.9.3 Overload resolution [temp.over]

When a call to the name of a function or function template is written (explicitly, or implicitly using the operator notation), template argument deduction (13.9.2) and checking of any explicit template arguments (13.3) 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 12.3.3. 143

2 [Example:

```c
  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 a, int b)
    char m2 = max(c,d); // max(char a, char b)
    int m3 = max(a,c);  // error: cannot generate max(int,char)
  }
```

Adding the non-template function

```c
  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]

3 [Example: Here is an example involving conversions on a function argument involved in template-argument deduction:

```c
  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(B<int>&di)
  }
```

143) 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 12.3.3 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.
4 [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]

5 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:

```cpp
template<class T> void f(T); // declaration

void g() {
    f("Annemarie");       // call of f<const char*>  
}
```

The call of \texttt{f} is well-formed even if the template \texttt{f} is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for \texttt{f<const char*>}, either implicitly or explicitly generated, is present in some translation unit. — end example]

13.10 Deduction guides [temp.deduct.guide]

1 Deduction guides are used when a \textit{template-name} appears as a type specifier for a deduced class type (9.1.7.6). Deduction guides are not found by name lookup. Instead, when performing class template argument deduction (12.3.1.8), any deduction guides declared for the class template are considered.

```latex
deduction-guide: explicit_opt template-name \{ parameter-declaration-clause \} \rightarrow simple-template-id ;
```

2 [Example:

```cpp
    template<class T, class D = int>
    struct S {
        T data;
    };

    template<class U>
    S(U) -> S<typename U::type>;

    struct A {
        using type = short;
        operator type();
    };

    S x(A());       // x is of type S<short, int>
```

— end example]

3 The same restrictions apply to the \textit{parameter-declaration-clause} of a deduction guide as in a function declaration (9.2.3.5). The \textit{simple-template-id} shall name a class template specialization. The \textit{template-name} shall be the same identifier as the \textit{template-name} of the \textit{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 \textit{parameter-declaration-clauses}. 

§ 13.10 417
14 Exception handling

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 [14.3].

2 A `try-block` is a statement (Clause 8). [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:

```c
void f() {
    goto l1; // ill-formed
    goto l2; // ill-formed
    try {
        goto l1; // OK
        goto l2; // ill-formed
        l1: ;
    } catch (...) {
        l2: ;
        goto l1; // ill-formed
        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:

```c
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 (6.8.1).

14.1 Throwing an exception [except.throw]

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 (6.6.5.4.1), dynamic_cast (7.6.1.6), typeid (7.6.1.7), new-expressions (7.6.2.5), 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 (14.3); “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 (9.3, 11.3.4.2) a temporary object, called the exception object. An lvalue denoting the temporary is used to initialize the variable declared in the matching handler (14.3). If the type of the exception object would be an incomplete type, an abstract class type (11.6.3), or a pointer to an incomplete type other than cv void the program is ill-formed.
The memory for the exception object is allocated in an unspecified way, except as noted in 6.6.5.4.1. 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:

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;
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 (6.8.2.1). [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 32.6. — 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 (11.9.5). The destructor is potentially invoked (11.3.6).

An exception is considered caught when a handler for that exception becomes active (14.3). [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 (14.5.2) directly invokes a function that exits via an exception, the function `std::terminate` is called (14.5.1). [Example:

```cpp
struct C {
    C() { }
    C(const C&) {
        if (std::uncaught_exceptions()) {
            throw 0; // throw during copy to handler's exception-declaration object (14.3)
        }
    }
};

int main() {
    try {
        throw C(); // calls std::terminate if construction of the handler's exception-declaration object is not elided (11.9.5)
    } catch(C) { }
}
```

— end example] [Note: Consequently, destructors should generally catch exceptions and not let them propagate. — end note]

### 14.2 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 (8.6.3), 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:

```cpp
struct A {
};

struct Y { -Y() noexcept(false) { throw 0; } };

A f() {
    try {
        A a;
        Y y;
        A b;
        return {};
    } catch(Y) { /* handler */ }
}
```

A function `f()` is declared, which throws an exception of type `std::exception`. The exception is caught by a handler that prints a message. The exception is then rethrown, and caught again by the same handler. The `catch` block contains a `return` statement, which is expected to cause the destructor of the exception object to be called. However, since the exception is caught again before the destructor is called, the `catch` block will not be executed. [Note: The order of the exceptions being caught and the corresponding destructors being called is specified in 6.8.2.1. — end note]
} catch (...) {  
}  
return {};  // #2

At #1, the returned object of type A is constructed. Then, the local variable b is destroyed (8.6). 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 (9.3) 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 (7.6.2.5), the matching deallocation function (6.6.5.4.2), if any, is called to free the storage occupied by the object. —end note]

### 14.3 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 cv 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:

```cpp
class Matherr { /* ... */ virtual void vf(); };  
class Overflow: public Matherr { /* ... */ };  
class Underflow: public Matherr { /* ... */ };  
class Zerodivide: public Matherr { /* ... */ };  

void f() {  
  try {  
    g();  
  } catch (...) {  
  }  
}
```]
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

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]

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.

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.

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.

The exception with the most recently activated handler that is still active is called the currently handled exception.

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 (14.5.1).

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.

The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function-try-block.

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 (6.8.3.1).

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.

If a return statement (8.6.3) appears in a handler of the function-try-block of a constructor, the program is ill-formed.

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 8.6.3).

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 (9.3) from the corresponding base class subobject of the exception object;

(15.2) — otherwise, the variable is copy-initialized (9.3) from the exception object.

The lifetime of the variable ends when the handler exits, after the destruction of any automatic objects initialized within the handler.

When the handler declares an object, any changes to that object will not affect the exception object. When the handler declares a reference to an object, any changes to the referenced object are changes to the exception object and will have effect should that object be rethrown.

14.4 Exception specifications

The predicate indicating whether a function cannot exit via an exception is called the exception specification of the function. If the predicate is false, the function has a potentially-throwing exception specification,
otherwise it has a **non-throwing exception specification**. The exception specification is either defined implicitly, or defined explicitly by using a **noexcept-specifier** as a suffix of a function declarator (9.2.3.5).

```
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 (9.3). 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 (13.8.2) 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:

```
struct B {
  virtual void f() noexcept;
  virtual void g();
  virtual void h() noexcept = delete;
};

struct D: B {
  void f(); // ill-formed
  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 (14.3) encounters the outermost block of a function with a non-throwing exception specification, the function `std::terminate` is called (14.5.1). [Note: An implementation shall not 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:

```
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

- 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
- 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 (6.8.1)) that is potentially-throwing, or
- e is a **throw-expression** (7.6.18), or
- e is a **dynamic_cast** expression that casts to a reference type and requires a runtime check (7.6.1.6), or
- 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 (6.8.1) of e is potentially-throwing.

An implicitly-declared constructor for a class X, or a constructor without a noexcept-specifier that is defaulted on its first declaration, has a potentially-throwing exception specification if and only if any of the following constructs is potentially-throwing:

- 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 (14.2), 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 (14.1, 14.5.1). — end note]

The exception specification for an implicitly-declared destructor, or a destructor without a noexcept-specifier, is potentially-throwing if and only if any of the destructors for any of its potentially constructed subobjects is potentially-throwing or the destructor is virtual and the destructor of any virtual base class is potentially-throwing.

The exception specification for an implicitly-declared assignment operator, or an assignment-operator without a noexcept-specifier that is defaulted on its first declaration, is potentially-throwing if and only if the invocation of any assignment operator in the implicit definition is potentially-throwing.

The exception specification for a comparison operator (7.6.8, 7.6.9, 7.6.10) without a noexcept-specifier that is defaulted on its first declaration is potentially-throwing if and only if the invocation of any comparison operator in the implicit definition is potentially-throwing.

[Example:

```c
struct A {
    A(int = (A(5), 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 Y(), 0)) noexcept;
    ~B() noexcept(false);
};
int n = 7;
struct D : public A, public B {
    // 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() potentially-throwing
};
```

Furthermore, if A::~A() were virtual, the program would be ill-formed since a function that overrides a virtual function from a base class shall not have a potentially-throwing exception specification if the base class function has a non-throwing exception specification. — end example]

An exception specification is considered to be needed when:

- in an expression, the function is the unique lookup result or the selected member of a set of overloaded functions (6.4, 12.3, 12.4);
- the function is odr-used (6.2) or, if it appears in an unevaluated operand, would be odr-used if the expression were potentially-evaluated;
— 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 special member 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 special member 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.

14.5 Special functions

The function std::terminate (14.5.1) 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.

14.5.1 The std::terminate function

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 (14.1), calls a function that exits via an exception, or
— when the exception handling mechanism cannot find a handler for a thrown exception (14.3), or
— when the search for a handler (14.3) encounters the outermost block of a function with a non-throwing exception specification (14.4), or
— when the destruction of an object during stack unwinding (14.2) terminates by throwing an exception, or
— when initialization of a non-local variable with static or thread storage duration (6.8.3.3) exits via an exception, or
— when destruction of an object with static or thread storage duration exits via an exception (6.8.3.4), or
— when execution of a function registered with std::atexit or std::at_quick_exit exits via an exception (17.5), or
— when evaluation of the predicate of a contract (9.11.4) exits via an exception, or
— when the violation handler invoked for a failed contract condition check (9.11.4) on a noexcept function exits via an exception, or
— when the violation handler has completed after a failed contract check and the continuation mode is off, or
— when a throw-expression (7.6.18) with no operand attempts to rethrow an exception and no exception is being handled (14.1), 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 (32.3.2.2), 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 copy assignment operator is invoked on an object of type std::thread that refers to a joinable thread (32.3.2.3, 32.3.2.4), or
— when a call to a wait(), wait_until(), or wait_for() function on a condition variable (32.5.3, 32.5.4) fails to meet a postcondition.
— end note]
In such cases, the function `std::terminate` is called (17.9.4). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate` is called. In the situation where the search for a handler (14.3) encounters the outermost block of a function with a non-throwing exception specification (14.4), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before the function `std::terminate` is called. In all other situations, the stack shall not be unwound before the function `std::terminate` is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to the function `std::terminate`.

### 14.5.2 The `std::uncaught_exceptions()` function

An exception is considered uncaught after completing the initialization of the exception object (14.1) until completing the activation of a handler for the exception (14.3). This includes stack unwinding. 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]

1 A preprocessing directive consists of a sequence of preprocessing tokens that satisfies the following constraints: The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) either is 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. The last token in the sequence is the first new-line character that follows the first token in the sequence.\(^{144}\) A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

\[
\text{preprocessing-file:}
\begin{align*}
\text{group}_\text{opt} \\
\text{group:} \\
& \text{group-part} \\
& \text{group group-part} \\
\text{group-part:} \\
& \text{control-line} \\
& \text{if-section} \\
& \text{text-line} \\
& \# \text{conditionally-supported-directive} \\
\text{control-line:} \\
& \# \text{include pp-tokens new-line} \\
& \# \text{define identifier replacement-list new-line} \\
& \# \text{define identifier lparen identifier-list}_\text{opt} ) \text{replacement-list new-line} \\
& \# \text{define identifier lparen identifier \ldots ) replacement-list new-line} \\
& \# \text{define identifier lparen identifier-list, } \ldots ) \text{replacement-list new-line} \\
& \# \text{undef identifier new-line} \\
& \# \text{line pp-tokens new-line} \\
& \# \text{error pp-tokens}_\text{opt new-line} \\
& \# \text{pragma pp-tokens}_\text{opt new-line} \\
& \# \text{new-line} \\
\text{if-section:} \\
& \text{if-group elif-groups}_\text{opt else-group}_\text{opt endif-line} \\
\text{if-group:} \\
& \# \text{if constant-expression new-line} \text{group}_\text{opt} \\
& \# \text{ifdef identifier new-line} \text{group}_\text{opt} \\
& \# \text{ifndef identifier new-line} \text{group}_\text{opt} \\
\text{elif-groups:} \\
& \text{elif-group} \\
& \text{elif-groups elseif-group} \\
\text{elif-group:} \\
& \# \text{elif constant-expression new-line} \text{group}_\text{opt} \\
\text{else-group:} \\
& \# \text{else new-line} \text{group}_\text{opt} \\
\text{endif-line:} \\
& \# \text{endif new-line} \\
\text{text-line:} \\
& \text{pp-tokens}_\text{opt new-line} \\
\text{conditionally-supported-directive:} \\
& \text{pp-tokens new-line} \\
\text{lparen:} \\
& \text{a ( character not immediately preceded by white-space)
\}

\(^{144}\) 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 15.5.2, for example).
A text line shall not begin with a # preprocessing token. A conditionally-supported-directive shall not begin with any of the directive names appearing in the syntax. A conditionally-supported-directive is conditionally-supported with implementation-defined semantics.

When in a group that is skipped (15.1), 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 introducing # preprocessing 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]

### 15.1 Conditional inclusion

```c
defined-macro-expression:
    defined identifier
    defined ( identifier )

h-preprocessing-token:
    any preprocessing-token other than >

h-pp-tokens:
    h-preprocessing-token
    h-pp-tokens h-preprocessing-token

header-name-tokens:
    string-literal
    < h-pp-tokens >

has-include-expression:
    __has_include ( header-name )
    __has_include ( header-name-tokens )

has-attribute-expression:
    __has_cpp_attribute ( pp-tokens )
```

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.

---

145) 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.
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 (15.4), 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 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 16. 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 16 — __has_cpp_attribute values

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>201806L</td>
</tr>
<tr>
<td>carries_dependency</td>
<td>200809L</td>
</tr>
<tr>
<td>deprecated</td>
<td>201309L</td>
</tr>
<tr>
<td>ensures</td>
<td>201806L</td>
</tr>
<tr>
<td>expects</td>
<td>201806L</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>201603L</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 groupopt
    # elif constant-expression new-line groupopt
```

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 and has-include-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 (8.4.1) 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.

13 Preprocessing directives of the forms

```c
#define identifier new-line group_opt
#define identifier new-line group_opt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to #if defined identifier and #if !defined identifier respectively.

14 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.146] [Example: This demonstrates a way to include a library optional facility only if it is available:]

```c
#include <optional>
#include <optional>
define have_optional 1
#endif
#endif
```

15 [Example: This demonstrates a way to use the attribute [[acme::deprecated]] only if it is available.]

```c
#define ATTR_DEPRECATED(msg) [[acme::deprecated(msg)]]
define ATTR_DEPRECATED(msg) [[deprecated(msg)]]
define ATTR_DEPRECATED("This function is deprecated") void anvil();
```

146) 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.
15.2 Source file inclusion  

A `#include` directive shall identify a header or source file that can be processed by the implementation.

A preprocessing directive of the form

```
#include <h-char-sequence> new-line
```

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

A preprocessing directive of the form

```
#include "q-char-sequence" new-line
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
#include <h-char-sequence> new-line
```

with the identical contained sequence (including `>` characters, if any) from the original directive.

A preprocessing directive of the form

```
#include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (i.e., each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined. 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), the preprocessing directive is instead replaced by the `preprocessing-tokens`

```
import header-name;
```

[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>
#define INCFILE "vers1.h"
#include INCFILE
```

—end example]
15.3 Global module fragment

\[cpp.glob.frag\]

\[
\text{pp-global-module-fragment:}
\]

\[
\text{module ; pp-balanced-token-seq module}
\]

1 If the first two preprocessing tokens at the start of phase 4 of translation are `module ;`, the result of preprocessing shall begin with a `pp-global-module-fragment` for which all preprocessing-tokens in the `pp-balanced-token-seq` were produced directly or indirectly by source file inclusion (15.2), and for which the second `module` preprocessing-token was not produced by source file inclusion or macro replacement (15.5). Otherwise, the first two preprocessing tokens at the end of phase 4 of translation shall not be `module ;`.

15.4 Header units

\[cpp.module\]

\[
\text{import-seq:}
\]

\[
\text{top-level-token-seq opt export opt import}
\]

\[
\text{top-level-token-seq:}
\]

\[
\text{any pp-balanced-token-seq ending in ; or } \}
\]

\[
pp-import:
\]

\[
\text{import header-name pp-import-suffix opt ;}
\]

\[
\text{import header-name-tokens pp-import-suffix opt ;}
\]

\[
pp-import-suffix:
\]

\[
\text{pp-import-suffix-token pp-import-suffix pp-import-suffix-token}
\]

\[
\text{pp-import-suffix-token:}
\]

\[
\text{any pp-balanced-token other than ;}
\]

\[
pp-balanced-token-seq:
\]

\[
\text{pp-balanced-token}
\]

\[
\text{pp-balanced-token-seq pp-balanced-token}
\]

\[
pp-balanced-token:
\]

\[
\text{pp-ldelim pp-balanced-token-seq opt pp-rdelim}
\]

\[
\text{any preprocessing-token other than a pp-ldelim or pp-rdelim}
\]

\[
pp-ldelim: \text{one of}
\]

\[
\{ \{ \text{<: <\%}
\]

\[
pp-rdelim: \text{one of}
\]

\[
\} \} \text{ :> %}
\]

1 A sequence of preprocessing-tokens matching the form of a `pp-import` instructs the preprocessor to import macros from the header unit (10.3) denoted by the `header-name`. A `pp-import` is only recognized when the sequence of tokens produced by phase 4 of translation up to the `import` token forms an import-seq, and the `import` token is not within the `header-name-tokens` or `pp-import-suffix` of another `pp-import`. The `;` preprocessing-token terminating a `pp-import` shall not have been produced by macro replacement (15.5). The point of macro import for a `pp-import` is immediately after the `;` terminating the `pp-import`.

2 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]

3 Each `#define` directive encountered when preprocessing each translation unit in a program results in a distinct macro definition. 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:

\[(3.1)\] 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).

\[(3.2)\] 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.
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.

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 (15.5). [Note: The relative
order of \textit{pp-imports} has no bearing on whether a particular macro definition is active. — end note]

\textbf{Example:}

Importable header "a.h"

\begin{verbatim}
#define X 123 // #1
#define Y 45  // #2
#define Z a  // #3
#undef X   // point of undefinition of #1 in "a.h"
\end{verbatim}

Importable header "b.h"

\begin{verbatim}
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
\end{verbatim}

Importable header "c.h"

\begin{verbatim}
#define Y 45  // #4
#define Z c  // #5
\end{verbatim}

Importable header "d.h"

\begin{verbatim}
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 #3 are valid redefinitions
int c = Z;   // error: active macro definitions #2 and #3 are not valid redefinitions of Z
\end{verbatim}

— end example]

\section{15.5 Macro replacement} \[cpp.replace\]

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 )
preprocessing token that terminates the invocation.

The identifiers \_\_VA_ARGS\_\_ and \_\_VA_OPT\_\_ shall occur only in the replacement-list of a function-like
macro that uses the ellipsis notation in the parameters.

A parameter identifier in a function-like macro shall be uniquely declared within its scope.

The identifier immediately following the \texttt{define} is called the \textit{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 \# 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 *object-like macro* that causes each subsequent instance of the macro name\textsuperscript{148} to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.\textsuperscript{149} The replacement list is then rescanned for more macro names as specified below.

A preprocessing directive of the form

\begin{verbatim}
# 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
\end{verbatim}

defines a *function-like macro* with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the `#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,\textsuperscript{150} the behavior is undefined.

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 ...).

### 15.5.1 Argument substitution

**va-opt-replacement:** \[cpp.subst\]

\[\__VA_OPT\__ \ ( \ pp-tokens_{opt} )\]

1. 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:

   (1.1) — 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.

   (1.2) — 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.

   **[Example:]**

   ```c
   #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;
   — end example
   ```

2. 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.

3. 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

\textsuperscript{148}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{149}An alternative token (5.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not possible to define a macro whose name is the same as that of an alternative token.

\textsuperscript{150}A *conditionally-supported-directive* is a preprocessing directive regardless of whether the implementation supports it.
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 (15.5.3,
15.5.4). 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 (15.5.2), and can be
removed by rescanning and further replacement (15.5.4). —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__ // ill-formed: ## may not appear at
// the beginning of a replacement list (15.5.3)
#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__() // ill-formed: # may not appear at
#define H5B(X) a ## X ## b
#define H5C(X) H5B(X)
#define H5C(H5A()) // replaced by ab
—end example]

15.5.2 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.
15.5.3 The ## operator

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

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.\footnote{Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.}

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 placemarkers 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.

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:

```
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]

15.5.4 Rescanning and further replacement

After all parameters in the replacement list have been substituted and # and ## processing has taken place, all placemaker 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.

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.

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 15.11 below.

15.5.5 Scope of macro definitions

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.

A preprocessing directive of the form

```
# undef identifier new-line
```

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

[Example: The simplest use of this facility is to define a “manifest constant”, as in

\[151\)
[Example: The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

#define max(a, b) ((a) > (b) ? (a) : (b))

The parentheses ensure that the arguments and the resulting expression are bound properly. —end example]

[Example: To illustrate the rules for redefinition and reexamination, the sequence

#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define m(a) a(w)
#define w 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)-(h 5) & m
(f)"m(m);
p() i[q()] = { q(1), r(2,3), r(4, ), r(5), r( ) };
char c[2][6] = { str(hello), str( ) };
results in
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))"m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", " " };
—end example]

[Example: To illustrate the rules for creating character string literals and concatenating tokens, the sequence

#define str(s) # s
#define xstr(s) str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \n 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"

debug(1, 2);
%fputs(str(strncmp("abc\0d", "abc", \4') == 0) str(: @n, s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
results in

printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
%fputs(strncmp("abc\0d", "abc", \4') == 0" : @n, s);
#include "vers2.h"  

"hello";
"hello" =, world"

or, after concatenation of the character string literals,

printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp("abc\0d", "abc", \"\d\") == 0: 0\n", s);

Space around the # and ## tokens in the macro definition is optional. — end example]

7 [Example: To illustrate the rules for placemarker preprocessing tokens, the sequence

#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,,),
          t(10,,), t(,,11), t(,,12), t(,,) };

results in

int j[] = { 123, 45, 67, 89,
10, 11, 12, }; — end example]

8 [Example: To demonstrate the redefinition rules, the following sequence is valid.

#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* white space */ (1-1) /* other */
#define FUNC_LIKE(a) ( a )
#define FUNC_LIKE(b) ( a ) /* note the white space */
#define FUNC_LIKE(b) ( b ) /* other stuff on this line */

But the following redefinitions are invalid:

#define OBJ_LIKE (0)  // different token sequence
#define OBJ_LIKE (1 - 1)  // different white space
#define FUNC_LIKE(a) ( a )  // different parameter usage
#define FUNC_LIKE(b) ( b )  // different parameter spelling

— end example]

9 [Example: Finally, to show the variable argument list macro facilities:

#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))

debug("Flag");
debug("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]

15.6 Line control [cpp.line]

1 The string literal of a #line directive, if present, shall be a character string literal.

2 The line number of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.2) while processing the source file to the current token.

3 A preprocessing directive of the form

    # line digit-sequence new-line
causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

4 A preprocessing directive of the form

```
# line digit-sequence * s-char-sequence_{opt} * new-line
```

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

5 A preprocessing directive of the form

```
# line pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

### 15.7 Error directive [cpp.error]

1 A preprocessing directive of the form

```
# error pp-tokens_{opt} new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

### 15.8 Pragma directive [cppPragma]

1 A preprocessing directive of the form

```
# pragma pp-tokens_{opt} 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.

### 15.9 Null directive [cpp.null]

1 A preprocessing directive of the form

```
# new-line
```

has no effect.

### 15.10 Predefined macro names [cpp.predefined]

1 The following macro names shall be defined by the implementation:

```
__cplusplus
```

The integer literal 201703L. [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).\(^{152}\)

```
__LINE__
```

The presumed line number (within the current source file) of the current source line (an integer literal).\(^{153}\)

\(^{152}\) The presumed source file name can be changed by the `#line` directive.

\(^{153}\) The presumed line number can be changed by the `#line` directive.
__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. (7.6.2.5). — 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 theasctime function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

The names listed in Table 17.
The macros defined in Table 17 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>
<tr>
<td>__cpp_capture_star_this</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_char8_t</td>
<td>201811L</td>
</tr>
<tr>
<td>__cpp_conditionalexplicit</td>
<td>201806L</td>
</tr>
<tr>
<td>__cpp_constexpr</td>
<td>201603L</td>
</tr>
<tr>
<td>__cpp_coroutines</td>
<td>201902L</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>201703L</td>
</tr>
<tr>
<td>__cpp_delegating_constructors</td>
<td>200604L</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>201304L</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 destroying delete</td>
<td>201806L</td>
</tr>
<tr>
<td>__cpp impl three way comparison</td>
<td>201711L</td>
</tr>
<tr>
<td>__cpp inheriting constructors</td>
<td>201511L</td>
</tr>
<tr>
<td>__cpp init captures</td>
<td>201304L</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 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>201411L</td>
</tr>
<tr>
<td>__cpp nontype template parameter_auto</td>
<td>201606L</td>
</tr>
<tr>
<td>__cpp nontype template parameter_class</td>
<td>201806L</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>
</tbody>
</table>
Table 17 — Feature-test macros (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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_variable_templates</td>
<td>201304L</td>
</tr>
<tr>
<td>__cpp_variadic_templates</td>
<td>200704L</td>
</tr>
<tr>
<td>__cpp_variadic_using</td>
<td>201611L</td>
</tr>
</tbody>
</table>

2 The following macro names are conditionally defined by the implementation:

   __STDC__
   Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.

   __STDC_MB_MIGHT_NEQ_WC__
   The integer literal 1, intended to indicate that, in the encoding for wchar_t, a member of the basic
   character set need not have a code value equal to its value when used as the lone character in an
   ordinary character literal.

   __STDC_VERSION__
   Whether __STDC_VERSION__ is predefined and if so, what its value is, are implementation-defined.

   __STDC_ISO_10646__
   An integer literal of the form yyyymmL (for example, 199712L). If this symbol is defined, then every
   character in the Unicode required set, when stored in an object of type wchar_t, has the same value
   as the code point of that character. The Unicode required set consists of all the characters that are
   defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year
   and month.

   __STDCPP_STRICT_POINTER_SAFETY__
   Defined, and has the value integer literal 1, if and only if the implementation has strict pointer
   safety (6.6.5.4.3).

   __STDCPP_THREADS__
   Defined, and has the value integer literal 1, if and only if a program can have more than one thread of
   execution (6.8.2).

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 defined, is the subject of a #define
or a #undef preprocessing directive, the behavior is undefined. Any other predefined macro names shall
begin with a leading underscore followed by an uppercase letter or a second underscore.

15.11 Pragma operator

1 A unary operator expression of the form:

Pragma ( string-literal )

[cppPragmaOp]
is processed as follows: The string literal is *destringized* by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence \" by a double-quote, and replacing each escape sequence \ by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

2 [Example:

```c
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```c
__Pragma ( "listing on "/..\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```c
#define LISTING(x) __Pragma(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )
```

— end example]
16 Library introduction

16.1 General

This Clause describes the contents of the C++ standard library, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

The following subclauses describe the 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 18.

Table 18 — 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 (7.6.2.5, 7.6.2.6) 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 (6.6.5.4), 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 `iostream` components that are the primary mechanism for C++ program input and output. They can be used with other elements of the library, particularly strings, locales, and iterators.

The regular expressions library (Clause 30) provides regular expression matching and searching.

The atomic operations library (Clause 31) allows more fine-grained concurrent access to shared data than is possible with locks.

The thread support library (Clause 32) provides components to create and manage threads, including mutual exclusion and interthread communication.

16.2 The C standard library

The C++ standard library also makes available the facilities of the C standard library, suitably adjusted to ensure static type safety.

The descriptions of many library functions rely on the C standard library for the semantics of those functions. In some cases, the signatures specified in this document may be different from the signatures in the C standard library, and additional overloads may be declared in this document, but the behavior and the preconditions (including any preconditions implied by the use of an ISO C `restrict` qualifier) are the same unless otherwise stated.

16.3 Definitions

[Note: Clause 3 defines additional terms used elsewhere in this document. — end note]

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

Object which, when treated sequentially, can represent text

[Note 1 to entry: The term does not mean only `char`, `char8_t`, `char16_t`, `char32_t`, and `wchar_t` objects, 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 (12.5) for any of the equality (7.6.10) or relational (7.6.9) 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 `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 `conditional-expression` CE (7.6.16) would not prevent CE from being a core constant expression (7.7)
16.3.7  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  default behavior
(implementation) specific behavior provided by the implementation, within the scope of the required behavior

16.3.9  default behavior
(specification) description of replacement function and handler function semantics

16.3.10  direct-non-list-initialization
direct-initialization (9.3) that is not list-initialization (9.3.4)

16.3.11  expression-equivalent
expressions that all have the same effects, either are all potentially-throwing (14.4) 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 \texttt{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  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  iostream class templates
templates, defined in Clause 29, that take two template arguments

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

16.3.14  modifier function
class member function (11.3.1) other than a constructor, assignment operator, or destructor that alters the
state of an object of the class

16.3.15  move assignment
assignment of an rvalue of some object type to a modifiable lvalue of the same type

16.3.16  move construction
direct-initialization of an object of some type with an rvalue of the same type

16.3.17  NTCTS
sequence of values that have character type that precede the terminating null character type value char\( T()\)

16.3.18  observer function
class member function (11.3.1) 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 \texttt{const} member functions (11.3.2.1). — end note]
16.3.19  [defns.prog.def.spec]
program-defined specialization
explicit template specialization or partial specialization that is not part of the C++ standard library and not defined by the implementation

16.3.20  [defns.prog.def.type]
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]

16.3.21  [defns.projection]
projection
(function object argument) transformation that an algorithm applies before inspecting the values of elements

[Example:
```
std::pair<int, std::string_view> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::ranges::less{}, [] (auto const& p) { return p.first; });
```
sorts the pairs in increasing order of their first members:
```
{{0, "baz"}, {1, "bar"}, {2, "foo"}}
```
—end example]

16.3.22  [defns.referenceable]
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]

16.3.23  [defns.replacement]
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 (6.5). —end note]

16.3.24  [defns.repositional.stream]
repositional stream
stream (described in Clause 29) that can seek to a position that was previously encountered

16.3.25  [defns.required.behavior]
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]

16.3.26  [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]
16.3.27 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 16.5.5.7. — end note]

16.3.28 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

16.3.29 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

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

16.4.1.1 Elements

Each library clause contains the following elements, as applicable:

(1.1) — Summary
(1.2) — Requirements
(1.3) — Detailed specifications
(1.4) — References to the C standard library

16.4.1.2 Summary

The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

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

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:

(1.1) — Template arguments
(1.2) — Derived classes
(1.3) — Containers, iterators, and algorithms that meet an interface convention or satisfy a concept

154) 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
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 satisfy 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 30) 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.

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 satisfied. [Example: The required < operator of the StrictTotallyOrdered 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 satisfies the concept.

A declaration may explicitly impose requirements through its associated constraints (13.4.2). When the associated constraints refer to a concept (13.6.8), 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):

1. **constructor(s) and destructor**
2. **copying, moving & assignment functions**
3. **comparison functions**
4. **modifier functions**
5. **observer functions**
6. **operators and other non-member functions**

Descriptions of function semantics contain the following elements (as appropriate):

1. **Requires:** the preconditions for calling the function.

---

155) Although in some cases the code given is unambiguously the optimum implementation.

156) 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.

157) 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 `Expects` element.
— *Constraints*: the conditions for the function’s participation in overload resolution (12.3). [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 (13.4.2). — 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 (Clause 9). 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 (13.4.2) and also define the function as deleted. — end example]

— *Expects*: the conditions (sometimes termed preconditions) that the function assumes to hold whenever it is called. [Example: An implementation might express such conditions via an attribute such as [[expects]] (9.11.4). However, some such conditions might not lend themselves to expression via code. — end example]

— *Effects*: the actions performed by the function.

— *Synchronization*: the synchronization operations (6.8.2) applicable to the function.

— *Ensures*: the conditions (sometimes termed observable results or postconditions) 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.

4 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*, *Expects*, *Effects*, *Synchronization*, *Ensures*, *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 (8.6.3) in the code sequence. If \( F \)’s semantics contains a *Throws*, *Ensures*, or *Complexity* element, then that supersedes any occurrences of that element in the code sequence.

5 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.

6 If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations.\(^\text{158}\)

7 Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.

8 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 [structure.see.also]

1 Paragraphs labeled “SEE ALSO” contain cross-references to the relevant portions of other standards (Clause 2).

16.4.2 Other conventions [conventions]

1 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).

\(^{158}\) 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 \textit{exposition only}.

The following function is 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>>)
{ return std::forward<T>(v); } // exposition only
```

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 defined for the purpose of exposition. The declaration of such a type is followed by a comment ending in \textit{exposition only}. [Example:

```cpp
namespace std {
extern "C" using some-handler = int(int, void*, double); // exposition only
}
```

The type placeholder \texttt{some-handler} can now be used to specify a function that takes a callback parameter with C language linkage. —end example]

16.4.2.2.3 Enumerated types

Several types defined in Clause 29 are \textit{enumerated types}. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.

The enumerated type \texttt{enumerated} can be written:

```cpp
enum enumerated
{ V0, V1, V2, V3, ... };
```

Here, the names \texttt{C0}, \texttt{C1}, etc. represent \textit{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 \textit{bitmask types}. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a \texttt{bitset} (20.9.2).

The bitmask type \texttt{bitmask} can be written:

```cpp
// For exposition only.
// \texttt{int\_type} is an integral type capable of representing all values of the bitmask type.
enum bitmask : int\_type {
    V0 = 1 \ll 0, V1 = 1 \ll 1, V2 = 1 \ll 2, V3 = 1 \ll 3, ...
};
```

---

159) Examples from 16.5.3 include: \texttt{Cpp1\_EqualityComparable}, \texttt{Cpp1\_LessThanComparable}, \texttt{Cpp1\_CopyConstructible}. Examples from 23.3 include: \texttt{Cpp1\_InputIterator}, \texttt{Cpp1\_ForwardIterator}.

160) Such as an integer type, with constant integer values (6.7.1).
inline constexpr bitmask C0(V0);
inline constexpr bitmask C1(V1);
inline constexpr bitmask C2(V2);
inline constexpr bitmask C3(V3);

constexpr bitmask operator&(bitmask X, bitmask Y) {
    return static_cast<bitmask>(static_cast<int_type>(X) & static_cast<int_type>(Y));
}
constexpr bitmask operator|(bitmask X, bitmask Y) {
    return static_cast<bitmask>(static_cast<int_type>(X) | static_cast<int_type>(Y));
}
constexpr bitmask operator^(bitmask X, bitmask Y) {
    return static_cast<bitmask>(static_cast<int_type>(X) ^ static_cast<int_type>(Y));
}
constexpr bitmask operator~(bitmask X) {
    return static_cast<bitmask>(~static_cast<int_type>(X));
}

bitmask& operator&=(bitmask& X, bitmask Y) {
    X = X & Y; return X;
}
bitmask& operator|=(bitmask& X, bitmask Y) {
    X = X | Y; return X;
}
bitmask& operator^=(bitmask& X, bitmask Y) {
    X = X ^ Y; return X;
}

Here, the names C0, C1, etc. represent bitmask elements for this particular bitmask type. All such elements have distinct, nonzero values such that, for any pair Ci and Cj where i ≠ j, Ci & Cj is nonzero and Ci & Cj is zero. Additionally, the value 0 is used to represent an empty bitmask, in which no bitmask elements are set.

The following terms apply to objects and values of bitmask types:

(4.1) — To set a value Y in an object X is to evaluate the expression X |= Y.
(4.2) — To clear a value Y in an object X is to evaluate the expression X &= ~Y.
(4.3) — The value Y is set in the object X if the expression X & Y is nonzero.

16.4.2.2.5 Character sequences [character.seq]

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

(1.1) — A letter is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.
(1.2) — The decimal-point character is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clause 17 through Clause 32 and Annex D by a period, '.', which is also its value in the "C" locale, but may change during program execution by a call to setlocale(int, const char*),161 or by a change to a locale object, as described in 28.3 and Clause 29.
(1.3) — A character sequence is an array object (9.2.3.4) A that can be declared as TA[N], where T is any of the types char, unsigned char, or signed char (6.7.1), 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 S that points to its first element.

161) declared in <clocale> (28.5).
16.4.2.2.5.1 Byte strings

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.\textsuperscript{162}

The length of an NTBS is the number of elements that precede the terminating null character. An empty NTBS has a length of zero.

The value of an NTBS is the sequence of values of the elements up to and including the terminating null character.

A static NTBS is an NTBS with static storage duration.\textsuperscript{163}

16.4.2.2.5.2 Multibyte strings

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.\textsuperscript{164}

A static NTMBS is an NTMBS with static storage duration.

16.4.2.2.6 Customization Point Object types

A customization point object is a function object (20.14) with a literal class type that interacts with program-defined types while enforcing semantic requirements on that interaction.

The type of a customization point object shall satisfy Semiregular (18.6).

All instances of a specific customization point object type shall be equal (18.2).

The type T of a customization point object shall satisfy Invocable<const T&, Args...> (18.7.2) when the types in Args... meet the requirements specified in that customization point object’s definition. When the types of Args... do not meet the customization point object’s requirements, T shall not have a function call operator that participates in overload resolution.

Each customization point object type constrains its return type to satisfy a particular concept.

[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 (6.4.2). 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 (13.6.6.2) or more constrained (13.4.4) to be used by a customization point object. —end note]

16.4.2.3 Functions within classes

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 (11.3.4.2, 11.3.5, 11.3.6). It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

16.4.2.4 Operators

In this library, whenever a declaration is provided for an operator!=, operator>, operator<, or operator>= for a type T, its requirements and semantics are as follows, unless explicitly specified otherwise.

bool operator!=(const T& x, const T& y);

Requires: Type T is Cpp17EqualityComparable (Table 23).
Returns: !(x == y).

bool operator>=(const T& x, const T& y);

Requires: Type T is Cpp17LessThanComparable (Table 24).
Returns: y < x.

\textsuperscript{162} Many of the objects manipulated by function signatures declared in <cstring> (21.5) 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.

\textsuperscript{163} A string literal, such as "abc", is a static NTBS.

\textsuperscript{164} 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.
bool operator<=(const T& x, const T& y);

Requires: Type T is Cpp17LessThanComparable (Table 24).

Returns: !(y < x).

bool operator>=(const T& x, const T& y);

Requires: Type T is Cpp17LessThanComparable (Table 24).

Returns: !(x < y).

16.4.2.5 Private members

Clause 17 through Clause 32 and Annex D do not specify the representation of classes, and intentionally omit specification of class members (11.3). 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:

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 describes the entities and macros defined in the C++ standard library. 16.5.1.2 lists the standard library headers and some constraints on those headers. 16.5.1.3 lists requirements for a freestanding implementation of the C++ standard library.

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).

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 19.

The facilities of the C standard library are provided in the additional headers shown in Table 20.

§ 16.5.1.2 453
Table 19 — C++ library headers

| `<algorithm>`  | `<forward_list>`  | `<new>`  | `<string_view>` |
| `<any>`        | `<fstream>`      | `<numeric>` | `<sstream>`    |
| `<array>`      | `<functional>`   | `<optional>` | `<syncstream>` |
| `<atomic>`     | `<future>`       | `<ostream>`  | `<system_error>` |
| `<bit>`        | `<initializer_list>` | `<queue>` | `<thread>`  |
| `<bitset>`     | `<iomanip>`      | `<random>`   | `<tuple>`   |
| `<charconv>`   | `<ios>`          | `<ranges>`   | `<typeindex>` |
| `<chrono>`     | `<iosfwd>`       | `<ratio>`    | `<typeinfo>` |
| `<codecvt>`    | `<iostream>`     | `<regex>`    | `<type_traits>` |
| `<compare>`    | `<iosstream>`    | `<scoped_allocator>` | `<unordered_map>` |
| `<complex>`    | `<iterator>`     | `<set>`      | `<unordered_set>` |
| `<concepts>`   | `<limits>`       | `<shared_mutex>` | `<utility>`   |
| `<condition_variable>` | `<list>` | `<span>` | `<valarray>` |
| `<contract>`   | `<locale>`       | `<sstream>`  | `<variant>`  |
| `<deque>`      | `<map>`          | `<stack>`    | `<vector>`   |
| `<exception>`  | `<memory>`       | `<stdexcept>` | `<version>` |
| `<execution>`  | `<memory_resource>` | `<streambuf>`        |            |
| `<filesystem>` | `<mutex>`        | `<string>`   |            |

Table 20 — C++ headers for C library facilities

| `<cassert>` | `<cfenv>` | `<climits>` | `<csetjmp>` | `<cstdlib>` | `<cstring>` | `<ctime>` | `<cwchar>` | `<cwctype>` |
| `<cctype>`   | `<cfloat>` | `<locale>`  | `<csignal>` | `<cstdarg>` | `<cstio>`   | `<cuchar>` | `<csignal>` | `<cwchar>` |
| `<cerrno>`   | `<cinttypes>` | `<cmath>`   | `<cstdio>`   | `<ctime>`   | `<cstring>` | `<cuchar>` | `<csignal>` | `<cwchar>` |

4 Except as noted in Clause 16 through Clause 32 and Annex D, the contents of each header `<name>h` is the same as that of the corresponding header `<name>h` as specified in the C standard library (Clause 2). In the C++ standard library, however, the declarations (except for names which are defined as macros in C) are within namespace scope (6.3.6) 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` (9.8).

5 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]

6 Names that are defined as functions in C shall be defined as functions in the C++ standard library.

7 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.

8 D.6, C standard library headers, describes the effects of using the `<name>h` (C header) form in a C++ program.

9 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>`.)

10 Table 21 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.

---

169) 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.

170) In particular, including the standard header `<iso646.h>` has no effect.

171) The `*.` 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 22.

#### Table 22 — C++ headers for freestanding implementations

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2</td>
<td><code>&lt;cstddef&gt;</code></td>
</tr>
<tr>
<td>17.3</td>
<td><code>&lt;cfloat&gt;</code>, <code>&lt;limits&gt;</code>, <code>&lt;climits&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;cstdlib&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>17.9</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>17.12</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>

The supplied version of the header `<cstdlib>` shall declare at least the functions `abort`, `atexit`, `at_quick_exit`, `exit`, and `quick_exit` (17.5). The other headers listed in this table shall meet the same requirements as for a hosted implementation.

### Using the library

#### 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

1 The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive (15.2).

2 A translation unit may include library headers in any order (Clause 5). Each may be included more than once, with no effect different from being included exactly once, except that the effect of including either `<cassert>` or `<assert.h>` depends each time on the lexically current definition of `NDEBUG`.¹⁷²

3 A translation unit shall include a header only outside of any declaration or definition, and shall include the header 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

1 Entities in the C++ standard library have external linkage (6.5). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (9.10).

2 Whether a name from the C standard library declared with external linkage has `extern "C"` or `extern "C++"` linkage is implementation-defined. It is recommended that an implementation use `extern "C++"` linkage for this purpose.¹⁷³

3 Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

4 See also replacement functions (16.5.4.6), 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

1 The template definitions in the C++ standard library refer to various named requirements whose details are set out in Tables 23–30. 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.

2 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 (9.3) if one of those signatures is called using the default argument (9.2.3.6).

### Table 23 — Cpp17EqualityComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>convertible to <code>bool</code></td>
<td><code>==</code> is an equivalence relation, that is, it has the following properties:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— For all a, a == a.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— If a == b, then b == a.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>— If a == b and b == c, then a == c.</td>
</tr>
</tbody>
</table>

¹⁷² This is the same as the C standard library.

¹⁷³ 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 24 — Cpp17LessThanComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>convertible to bool</td>
<td>( \times ) is a strict weak ordering relation (25.7)</td>
</tr>
</tbody>
</table>

### Table 25 — Cpp17DefaultConstructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{T t;} )</td>
<td>object ( t ) is default-initialized</td>
</tr>
<tr>
<td>( \text{T u;} )</td>
<td>object ( u ) is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td>( \text{T() } )</td>
<td>an object of type ( T ) is value-initialized or aggregate-initialized</td>
</tr>
</tbody>
</table>

### Table 26 — Cpp17MoveConstructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{T u = rv;} )</td>
<td>( u ) is equivalent to the value of ( rv ) before the construction</td>
</tr>
<tr>
<td>( \text{T(rv)} )</td>
<td>( T(rv) ) is equivalent to the value of ( rv ) before the construction</td>
</tr>
</tbody>
</table>

\( rv \)’s state is unspecified [Note: \( rv \) must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether \( rv \) has been moved from or not. \( \text{—end note} \) ]

### Table 27 — Cpp17CopyConstructible requirements (in addition to Cpp17MoveConstructible)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{T u = v;} )</td>
<td>the value of ( v ) is unchanged and is equivalent to ( u )</td>
</tr>
<tr>
<td>( \text{T(v)} )</td>
<td>the value of ( v ) is unchanged and is equivalent to ( T(v) )</td>
</tr>
</tbody>
</table>

### Table 28 — 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>( t = rv )</td>
<td>T&amp;</td>
<td>( t )</td>
<td>If ( t ) and ( rv ) do not refer to the same object, ( t ) is equivalent to the value of ( rv ) before the assignment</td>
</tr>
</tbody>
</table>

\( rv \)’s state is unspecified. [Note: \( rv \) must still meet the requirements of the library component that is using it, whether or not \( t \) and \( rv \) refer to the same object. The operations listed in those requirements must work as specified whether \( rv \) has been moved from or not. \( \text{—end note} \) ]

### Table 29 — 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 30 — Cpp17Destructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u )\text{.=T()}</td>
<td>All resources owned by ( u ) are reclaimed, no exception is propagated.</td>
</tr>
</tbody>
</table>
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:

1. The expressions \( \text{swap}(t, u) \) and \( \text{swap}(u, t) \) are valid when evaluated in the context described below, and
2. These expressions have the following effects:
   1. The object referred to by \( t \) has the value originally held by \( u \) and
   2. The object referred to by \( u \) has the value originally held by \( t \).

The context in which \( \text{swap}(t, u) \) and \( \text{swap}(u, t) \) are evaluated shall ensure that a binary non-member function named “swap” is selected via overload resolution (12.3) on a candidate set that includes:

1. The two \text{swap} function templates defined in <utility> (20.2) and
2. The lookup set produced by argument-dependent lookup (6.4.2).

[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 \text{std::swap}(t, u) \) or \text{std::swap}(u, t) as appropriate. — end note]

[Note: It is unspecified whether a library component that has a swappable requirement includes the header <utility> to ensure an appropriate evaluation context. — end note]

An rvalue or lvalue \( t \) is swappable if and only if \( t \) is swappable with any rvalue or lvalue, respectively, of type \( T \).

A type \( X \) satisfying any of the iterator requirements (23.3) satisfies the \text{Cpp17ValueSwappable} requirements if, for any dereferenceable object \( x \) of type \( X \), \(*x\) is swappable.

[Example: User code can ensure that the evaluation of \text{swap} calls is performed in an appropriate context under the various conditions as follows:

```cpp
#include <utility>

// Requires: std::forward<T>(t) shall be swappable with std::forward<U>(u).
// for rvalues and lvalues

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
}

// 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 conditions for fundamental types
    }

    void swap(Proxy p, A& 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

A Cpp17NullablePointer type is a pointer-like type that supports null values. A type P satisfies the Cpp17NullablePointer requirements if:

1. P satisfies the Cpp17EqualityComparable, Cpp17DefaultConstructible, Cpp17CopyConstructible, Cpp17CopyAssignable, and Cpp17Destructible requirements,
2. lvalues of type P are swappable (16.5.3.2),
3. the expressions shown in Table 31 are valid and have the indicated semantics, and
4. P satisfies all the other requirements of this subclause.

A value-initialized object of type P produces the null value of the type. The null value shall be equivalent only to itself. A default-initialized object of type P may have an indeterminate value. [Note: Operations involving indeterminate values may cause undefined behavior. — end note]

An object p of type P can be contextually converted to bool (7.3). The effect shall be as if p != nullptr had been evaluated in place of p.

No operation which is part of the Cpp17NullablePointer requirements shall exit via an exception.

In Table 31, 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 31 — Cpp17NullablePointer requirements

<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>Ensures: u == nullptr</td>
</tr>
<tr>
<td>P u = np;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P(np)</td>
<td></td>
<td>Ensures: P(np) == nullptr</td>
</tr>
<tr>
<td>t = np P&amp;</td>
<td></td>
<td>Ensures: 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>contextually convertible to bool</td>
<td>!(a == np)</td>
</tr>
<tr>
<td>np != a</td>
<td>contextually convertible to bool</td>
<td>!(a == np)</td>
</tr>
</tbody>
</table>

16.5.3.4  Cpp17Hash requirements

A type H meets the Cpp17Hash requirements if:

1. it is a function object type (20.14),
2. it satisfies the Cpp17CopyConstructible (Table 27) and Cpp17Destructible (Table 30) requirements, and
3. the expressions shown in Table 32 are valid and have the indicated semantics.

Given Key is an argument type for function objects of type H, in Table 32 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.

16.5.3.5  Cpp17Allocator 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.

§ 16.5.3.5 459
Table 32 — *Cpp17Hash* requirements

<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>

The class template `allocator_traits` (20.10.9) supplies a uniform interface to all allocator types. Table 33 describes the types manipulated through allocators. Table 34 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 34 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 33 and 34, the use of `move` and `forward` always refers to `std::move` and `std::forward`, respectively.

Table 33 — Descriptive variable definitions

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T, U, C</td>
<td>any cv-unqualified object type (6.7)</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 <code>allocator_traits&lt;X&gt;</code></td>
</tr>
<tr>
<td>YY</td>
<td>the type <code>allocator_traits&lt;Y&gt;</code></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>

Table 34 — *Cpp17Allocator* requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::pointer</code></td>
<td>T*</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>X::const_pointer</code></td>
<td>X::pointer is convertible to X::const_pointer</td>
<td>pointer_traits&lt;X&gt;::pointer::rebind&lt;const T&gt;</td>
<td></td>
</tr>
<tr>
<td>§16.5.3.5</td>
<td></td>
<td></td>
<td>460</td>
</tr>
<tr>
<td><code>X::void_pointer</code></td>
<td>X::pointer is convertible to X::const_pointer</td>
<td>pointer_traits&lt;X&gt;::pointer::rebind&lt;volatile T&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Table 34 — 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::const_void_pointer</td>
<td>X::pointer, Y::const_void_pointer</td>
<td>X::const_pointer, and X::void_pointer are convertible to X::const_void_pointer. X::const_void_pointer and Y::const_void_pointer are the same type.</td>
<td>*p T&amp;</td>
</tr>
<tr>
<td>X::size_type</td>
<td>unsigned integer type</td>
<td>a type that can represent the size of the largest object in the allocation model.</td>
<td>*p</td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integer type</td>
<td>a type that can represent the difference between any two pointers in the allocation model.</td>
<td>p-&gt;m</td>
</tr>
<tr>
<td>typename typename</td>
<td>Y</td>
<td>For all U (including T), Y::template rebind&lt;T&gt;::other is X.</td>
<td>q-&gt;m</td>
</tr>
<tr>
<td>X::template rebind&lt;U&gt;::other</td>
<td>X::pointer</td>
<td>static_cast&lt;X::pointer&gt;(w) == p</td>
<td>static_cast&lt;X::pointer&gt;(w) == p</td>
</tr>
<tr>
<td>static_cast&lt;X::pointer&gt;(w)</td>
<td>X::pointer</td>
<td>static_cast&lt;X::pointer&gt;(x) == q</td>
<td>static_cast&lt;X::pointer&gt;(x) == q</td>
</tr>
<tr>
<td>static_cast&lt;X::const_pointer&gt;(x)</td>
<td>X::const_pointer</td>
<td>static_cast&lt;X::const_pointer&gt;(y) == p</td>
<td>static_cast&lt;X::const_pointer&gt;(y) == p</td>
</tr>
<tr>
<td>pointer_traits&lt;X::pointer_to&gt;(r)</td>
<td>X::pointer</td>
<td>same as p</td>
<td>same as p</td>
</tr>
<tr>
<td>a.allocate(n)</td>
<td>X::pointer</td>
<td>Memory is allocated for n objects of type T but objects are not constructed. allocate may throw an appropriate exception.174 [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>
</tbody>
</table>

174 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 34 — *Cpp17Allocator* requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.deallocate(p,n)</td>
<td>(not used)</td>
<td>Requires: p shall be a value returned by an earlier call to allocate that has not been invalidated by an intervening call to deallocate. n shall match the value passed to allocate 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 X::allocate()</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. operator== 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);</td>
<td></td>
<td>Shall not exit via an exception. Ensures: u == a</td>
<td></td>
</tr>
<tr>
<td>X u = a;</td>
<td></td>
<td>Shall not exit via an exception. Ensures: Y(u) == b, u == X(b)</td>
<td></td>
</tr>
<tr>
<td>X u(b);</td>
<td></td>
<td>Shall not exit via an exception. Ensures: The value of a is unchanged and is equal to u.</td>
<td></td>
</tr>
<tr>
<td>X u(std::move(a));</td>
<td></td>
<td>Shall not exit via an exception. Ensures: u is equal to the prior value of X(b).</td>
<td></td>
</tr>
<tr>
<td>X u = std::move(a);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u(std::move(b));</td>
<td></td>
<td></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::new ((void*)c) C(forward&lt;Args&gt;(args)...)</td>
<td></td>
</tr>
<tr>
<td>a.destroy(c)</td>
<td>(not used)</td>
<td>Effects: Destroys the object at c c-&gt;~C()</td>
<td></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>
</tbody>
</table>

§ 16.5.3.5
Table 34 — Cpp17Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::propagate_on_container_swap</code></td>
<td>Identical to or derived from <code>true_type</code> or <code>false_type</code></td>
<td><code>true_type</code> only if an allocator of type <code>X</code> should be swapped when the client container is swapped. See Note B, below.</td>
<td><code>false_type</code></td>
</tr>
<tr>
<td><code>X::is_always_equal</code></td>
<td>Identical to or derived from <code>true_type</code> or <code>false_type</code></td>
<td><code>true_type</code> only if the expression <code>a1 == a2</code> is guaranteed to be true for any two (possibly const) values <code>a1</code>, <code>a2</code> of type <code>X</code>.</td>
<td><code>true_type</code></td>
</tr>
</tbody>
</table>

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<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.

Note B: If `X::propagate_on_container_copy_assignment::value` is `true`, `X` shall satisfy the `Cpp17CopyAssignable` requirements (Table 29) and the copy operation shall not throw exceptions. If `X::propagate_on_container_move_assignment::value` is `true`, `X` shall satisfy the `Cpp17MoveAssignable` requirements (Table 28) and the move operation shall not throw exceptions. If `X::propagate_on_container_swap::value` is `true`, lvalues of type `X` shall be swappable (16.5.3.2) and the swap operation shall not throw exceptions.

An allocator type `X` shall meet the `Cpp17CopyConstructible` requirements (Table 27). The `X::pointer`, `X::const_pointer`, `X::void_pointer`, and `X::const_void_pointer` types shall meet the `Cpp17NullablePointer` requirements (Table 31). 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`, `addressof(*(a + n)) == addressof(*a) + n` is true.

Let `x1` and `x2` denote objects of (possibly different) types `X::void_pointer`, `X::const_void_pointer`, `X::pointer`, or `X::const_pointer`. Then, `x1` and `x2` are equivalently-valued pointer values, if and only if both `x1` and `x2` can be explicitly converted to the two corresponding objects `px1` and `px2` of type `X::const_pointer`, using a sequence of static_casts using only these four types, and the expression `px1 == px2` evaluates to true.

Let `w1` and `w2` denote objects of type `X::void_pointer`. Then for the expressions

- `w1 == w2`
- `w1 != w2`

either or both objects may be replaced by an equivalently-valued object of type `X::const_void_pointer` with no change in semantics.

Let `p1` and `p2` denote objects of type `X::pointer`. Then for the expressions

- `p1 == p2`
- `p1 != p2`
- `p1 < p2`
- `p1 <= p2`
- `p1 > p2`
- `p1 >= p2`
- `p1 - p2`

either or both objects may be replaced by an equivalently-valued object of type `X::const_pointer` with no change in semantics.
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 satisfies the requirements of Table 34:

```cpp
template<classTp>
structSimpleAllocator{
    typedefTpvalue_type;
    SimpleAllocator(ctorargs);

template<classT>SimpleAllocator(constSimpleAllocator<T>&other);

    [[nodiscard]]Tp*allocate(std::size_tn);
    voiddeallocate(Tp*p,std::size_tn);
};
template<classT,classU>
booloperator==(constSimpleAllocator<T>&,constSimpleAllocator<U>&);
template<classT,classU>
booloperator!=(constSimpleAllocator<T>&,constSimpleAllocator<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 satisfies the allocator completeness requirements if, whether or not $T$ is a complete type:

(1.1) $X$ is a complete type, and

(1.2) 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 [namespace.constraints]

16.5.4.2.1 Namespace `std` [namespace.std]

Unless otherwise specified, the behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std`.

Unless explicitly prohibited, a program may add a template specialization for any standard library class template to namespace `std` provided that (a) the added declaration depends on at least one program-defined type and (b) the specialization meets the standard library requirements for the original template.175

The behavior of a C++ program is undefined if it declares an explicit or partial specialization of any standard library variable template, except where explicitly permitted by the specification of that variable template.

The behavior of a C++ program is undefined if it declares

(4.1) an explicit specialization of any member function of a standard library class template, or

(4.2) an explicit specialization of any member function template of a standard library class or class template, or

175) 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.
— an explicit or partial specialization of any member class template of a standard library class or class
template, or
— a deduction guide for any standard library class template.

A program may explicitly instantiate a class template defined in the standard library only if the declaration
(a) depends on the name of at least one program-defined type and (b) the instantiation meets the standard
library requirements for the original template.

Let \( F \) denote a standard library function (16.5.5.4), 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 \( F \). \[ Note: Possible means of forming such pointers include application of the unary &
operator (7.6.2.1), addressof (20.10.11.2), or a function-to-pointer standard conversion (7.3.3). — end note \]
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.

Other than in namespace std or in a namespace within namespace std, a program may provide an overload for
any 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. \[ Note: This permits a (qualified or unqualified) call to the customization point
to invoke the most appropriate overload for the given arguments. — end note \]

A translation unit shall not declare namespace std to be an inline namespace (9.7.1).

### 16.5.4.2.2 Namespace posix

The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a
namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by
ISO/IEC 9945 and other POSIX standards.

### 16.5.4.2.3 Namespaces for future standardization

Top level namespaces with a name starting with std and followed by a non-empty sequence of digits are
reserved for future standardization. The behavior of a C++ program is undefined if it adds declarations
or definitions to such a namespace. \[ Example: The top level namespace std2 is reserved for use by future
revisions of this International Standard. — end example \]

### 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. bind1st,
6. bind2nd,

---

176) 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 (6.8.1), 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.
©ISO/IEC N4810

(1.7) — binder1st,
(1.8) — binder2nd,
(1.9) — const_mem_fun1_ref_t,
(1.10) — const_mem_fun1_t,
(1.11) — const_mem_fun_ref_t,
(1.12) — const_mem_fun_t,
(1.13) — get_temporary_buffer,
(1.14) — get_unexpected,
(1.15) — gets,
(1.16) — is_literal_type,
(1.17) — is_literal_type_v,
(1.18) — mem_fun1_ref_t,
(1.19) — mem_fun1_t,
(1.20) — mem_fun_ref_t,
(1.21) — mem_fun_ref,
(1.22) — mem_fun_t,
(1.23) — mem_fun,
(1.24) — not1,
(1.25) — not2,
(1.26) — pointer_to_binary_function,
(1.27) — pointer_to_unary_function,
(1.28) — ptr_fun,
(1.29) — random_shuffle,
(1.30) — raw_storage_iterator,
(1.31) — result_of,
(1.32) — result_of_t,
(1.33) — return_temporary_buffer,
(1.34) — set_unexpected,
(1.35) — unary_function,
(1.36) — unary_negate,
(1.37) — uncaugh_exception,
(1.38) — unexpected, and
(1.39) — unexpected_handler.

2 The following names are reserved as member types for previous standardization, and may not be used as a name for object-like macros in portable code:

(2.1) — argument_type,
(2.2) — first_argument_type,
(2.3) — io_state,
(2.4) — open_mode,
(2.5) — second_argument_type, and
(2.6) — seek_dir.

3 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, to the `attribute-tokens` described in 9.11, or to the identifiers `expects` or `ensures`, except that the names `likely` and `unlikely` may be defined as function-like macros (15.5).

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 name 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 (12.5.8) 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 (15.2), the behavior is undefined.

16.5.4.5 Derived classes

1 Virtual member function signatures defined for a base class in the C++ standard library may be overridden in a derived class defined in the program (11.6.2).

16.5.4.6 Replacement functions

1 Clause 17 through Clause 32 and Annex D describe the behavior of numerous functions defined by the C++ standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program (16.3).

2 A C++ program may provide the definition for any of the following dynamic memory allocation function signatures declared in header `<new>` (6.6.5.4, 17.6):

```c
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&)
```

177) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.

178) The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>`, and `va_end(va_list)`, declared or defined in `<cstdarg>`,<ref>179) The function signatures declared in `<cuchar>`, `<cwchar>`, and `<cwctype>` are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.</ref>

180) These types are `clock_t`, `div_t`, `FILE`, `fpos_t`, `fread_t`, `fseek_t`, `ftime_t`, `ldiv_t`, `mbstate_t`, `mtimdifft_t`, `sig_atomic_t`, `size_t`, `time_t`, `tm`, `va_list`, `wctrans_t`, `wctype_t`, and `wint_t`. 

§ 16.5.4.6 467
The program’s definitions are used instead of the default versions supplied by the implementation (17.6).
Such replacement occurs prior to program startup (6.2, 6.8.3). 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):

(1.1) — terminate_handler

A C++ program may install different handler functions during execution, by supplying a pointer to a function
defined in the program or the library as an argument to (respectively):

(2.1) — set_new_handler

(2.2) — set_terminate

See also subclauses 17.6.3, 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:

(3.1) — get_new_handler

(3.2) — get_terminate

Calling the set_* and get_* functions shall not incur a data race. A call to any of the set_* functions shall
synchronize with subsequent calls to the same set_* function and to the corresponding get_* function.

16.5.4.8 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++ pro-
gram. If these components do not meet their requirements, this document places no requirements on the
implementation.

In particular, the effects are undefined in the following cases:

(2.1) — for replacement functions (17.6.2), if the installed replacement function does not implement the semantics
of the applicable Required behavior: paragraph.

(2.2) — 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

(2.3) — for types used as template arguments when instantiating a template component, if the operations on
the type do not implement the semantics of the applicable Requirements subclause (16.5.3.5, 22.2, 23.3,
25.2, 26.2). Operations on such types can report a failure by throwing an exception unless otherwise
specified.

(2.4) — if any replacement function or handler function or destructor operation exits via an exception, unless
specifically allowed in the applicable Required behavior: paragraph.

(2.5) — if an incomplete type (6.7) 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. 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.

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.

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 (13.9.2.1) 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 16.5.5.9. [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]

2. If an object of a standard library type is accessed, and the beginning of the object’s lifetime (6.6.3) 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 Requires paragraph

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.

2. Violation of any preconditions specified in a function’s Expects: element results in undefined behavior.

16.5.4.12 Semantic requirements

If the semantic requirements of a declaration’s constraints (16.4.1.3) are not satisfied 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.

2. 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 16.5.5.9, access specifiers in 16.5.5.10, class derivation in 16.5.5.11, and exceptions in 16.5.5.12.

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.

2. 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 (6.2).

3. The C standard library headers (D.6) 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 (9.1.6).

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. 181

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 (6.4.2). [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 (9.1.6).

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 Constexpr functions and constructors

This document explicitly requires that certain standard library functions are constexpr (9.1.5). 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.

16.5.5.7 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).

16.5.5.8 Reentrancy

Except where explicitly specified in this document, it is implementation-defined which functions in the C++ standard library may be recursively reentered.

181) 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.
16.5.5.9 Data race avoidance

This subclause specifies requirements that implementations shall meet to prevent data races (6.8.2). 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 (6.8.2) 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 (6.8.2) 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 a static object 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]

4 A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it. [Note: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. — end note]

5 Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

6 Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible (6.8.2) to users.

[Note: This allows implementations to parallelize operations if there are no visible side effects. — end note]

16.5.5.10 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.

16.5.5.11 Derived classes

An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

Certain classes defined in the C++ standard library are required to be derived from other classes in the C++ standard library. An implementation may derive such a class directly from the required base or indirectly through a hierarchy of base classes with names reserved to the implementation.

In any case:

(3.1) — Every base class described as virtual shall be virtual;
(3.2) — Every base class not specified as virtual shall not be virtual;
(3.3) — Unless explicitly stated otherwise, types with distinct names shall be distinct types. 182

4 All types specified in the C++ standard library shall be non-final types unless otherwise specified.

16.5.5.12 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 handle for the base type.

Functions from the C standard library shall not throw exceptions 183 except when such a function calls a program-supplied function that throws an exception. 184

182) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as size_t (17.2) and streamoff (29.5.2).
183) 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.
184) The functions qsort() and bsearch() (25.10) meet this condition.
Destructor operations defined in the C++ standard library shall not throw exceptions. Every destructor in the C++ standard library shall behave as if it had a non-throwing exception specification.

Functions defined in the C++ standard library that do not have a *Throws:* paragraph but do have a potentially-throwing exception specification may throw implementation-defined exceptions.\(^{185}\) Implementations should report errors by throwing exceptions of or derived from the standard exception classes (17.6.3.1, 17.9, 19.2).

An implementation may strengthen the exception specification for a non-virtual function by adding a non-throwing exception specification.

### 16.5.5.13 Restrictions on storage of pointers

Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type `std::intptr_t` shall store such values in a traceable pointer location (6.6.5.4.3). [Note: Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. 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. — end note]

### 16.5.5.14 Value of error codes

Certain functions in the C++ standard library report errors via a `std::error_code` (19.5.3.1) object. That object’s `category()` member shall return `std::system_category()` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories. [Example: For operating systems that are based on POSIX, implementations should define the `std::system_category()` values as identical to the POSIX `errno` values, with additional values as defined by the operating system’s documentation. Implementations for operating systems that are not based on POSIX should define values identical to the operating system’s values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. — end example]

### 16.5.5.15 Moved-from state of library types

Objects of types defined in the C++ standard library may be moved from (11.3.4.2). 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.

\(^{185}\) In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc` (17.6.3.1).
17 Language support library
[language.support]

17.1 General
[support.general]

1 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.

2 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 contract violation handling, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 35.

Table 35 — Language support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.2 Common definitions</td>
<td>&lt;cstddef&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>17.3 Implementation properties</td>
<td>&lt;limits&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;climits&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cfloat&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;version&gt;</td>
</tr>
<tr>
<td>17.4 Integer types</td>
<td>&lt;cstdint&gt;</td>
</tr>
<tr>
<td>17.5 Start and termination</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>17.6 Dynamic memory management</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>17.7 Type identification</td>
<td>&lt;typeinfo&gt;</td>
</tr>
<tr>
<td>17.8 Contract violation handling</td>
<td>&lt;contract&gt;</td>
</tr>
<tr>
<td>17.9 Exception handling</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td>17.10 Initializer lists</td>
<td>&lt;initializer_list&gt;</td>
</tr>
<tr>
<td>17.11 Comparisons</td>
<td>&lt;compare&gt;</td>
</tr>
<tr>
<td>17.12 Coroutines</td>
<td>&lt;coroutine&gt;</td>
</tr>
<tr>
<td>17.13 Other runtime support</td>
<td>&lt;csignal&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;csetjmp&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdarg&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

17.2 Common definitions
[support.types]

17.2.1 Header <cstddef> synopsis
[cstddef.syn]

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;

§ 17.2.1
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;

}  // namespace std

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 offsetof(P, D) 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);

}  // namespace std

1 The contents and meaning of the header <cstdlib> are the same as the C standard library header <stddef.h>, except that it does not declare the type wchar_t, that it also declares the type byte and its associated operations (17.2.5), and as noted in 17.2.3 and 17.2.4.

See also:  ISO C 7.19

§ 17.2.2 synonym
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 mblen(const char* s, size_t n);
int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);

// 25.10, C standard library algorithms
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);

// 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 numer, int denom);
ldiv_t div(long long int numer, long long int denom); // see 16.2
ldiv_t div(long long int numer, long long int denom); // see 16.2
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);
}

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, 25.10, 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]

1 The type nullptr_t is a synonym for the type of a nullptr expression, and it has the characteristics described in 6.7.1 and 7.3.11. [Note: Although nullptr’s address cannot be taken, the address of another nullptr_t object that is an lvalue can be taken. — end note]

2 The macro NULL is an implementation-defined null pointer constant.¹⁸⁶

¹⁸⁶) Possible definitions include 0 and 0L, but not (void*)0.
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 (11.1) is conditionally-supported.\(^{187}\)

The expression `offsetof(type, member-designator)` is never type-dependent (13.7.2.2) and it is value-dependent (13.7.2.3) if and only if `type` is dependent. The result of applying the `offsetof` macro to a static data member or a function member is undefined. No operation invoked by the `offsetof` macro shall throw an exception and `noexcept(offsetof(type, member-designator))` shall be true.

The type `ptrdiff_t` is an implementation-defined signed integer type that can hold the difference of two subscripts in an array object, as described in 7.6.6.

The type `size_t` is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object (7.6.2.4).

[Note: It is recommended that implementations choose types for `ptrdiff_t` and `size_t` whose integer conversion ranks (6.7.4) are no greater than that of `signed long int` unless a larger size is necessary to contain all the possible values. —end note]

The type `max_align_t` is a trivial standard-layout type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context (6.6.6).

### § 17.2.5 byte type operations

```cpp
template<class IntType>
constexpr byte& operator<<=(byte& b, IntType shift) noexcept;

Constraints: `is_integral_v<IntType>` is true.

Effects: Equivalent to: `return b = b << shift;`

template<class IntType>
constexpr byte operator<<(byte b, IntType shift) noexcept;

Constraints: `is_integral_v<IntType>` is true.

Effects: Equivalent to:
```
return static_cast<byte>(static_cast<unsigned char>((
    static_cast<unsigned int>(b) << shift));
```

template<class IntType>
constexpr byte& operator>>=(byte& b, IntType shift) noexcept;

Constraints: `is_integral_v<IntType>` is true.

Effects: Equivalent to: `return b >> shift;`

template<class IntType>
constexpr byte operator>>(byte b, IntType shift) noexcept;

Constraints: `is_integral_v<IntType>` is true.

Effects: Equivalent to:
```
return static_cast<byte>(static_cast<unsigned char>((
    static_cast<unsigned int>(b) >> shift));
```

constexpr byte& operator|=(byte& l, byte r) noexcept;

Effects: Equivalent to: `return l = l | r;`

constexpr byte operator|(byte l, byte r) noexcept;

Effects: Equivalent to:
```
return static_cast<byte>(static_cast<unsigned char>(static_cast<unsigned int>(l) |
    static_cast<unsigned int>(r)));
```

\(^{187}\) Note that `offsetof` is required to work as specified even if unary operator is overloaded for any of the types involved.
constexpr byte& operator&=(byte& l, byte r) noexcept;

**Effects:** Equivalent to: return l = l & r;

constexpr byte operator&(byte l, byte r) noexcept;

**Effects:** Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(static_cast<unsigned int>(l) &
static_cast<unsigned int>(r)));
```

constexpr byte& operator^=(byte& l, byte r) noexcept;

**Effects:** Equivalent to:

```
return l = l ^ r;
```

constexpr byte operator^(byte l, byte r) noexcept;

**Effects:** Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(static_cast<unsigned int>(l) ^
static_cast<unsigned int>(r)));
```

constexpr byte operator~(byte b) noexcept;

**Effects:** Equivalent to:

```
return static_cast<byte>(static_cast<unsigned char>(
~static_cast<unsigned int>(b)));
```

template<class IntType>
constexpr IntType to_integer(byte b) noexcept;

**Constraints:** is_integral_v<IntType> is true.

**Effects:** Equivalent to: return static_cast<IntType>(b);

### 17.3 Implementation properties

**[support.limits]**

#### 17.3.1 General

**[support.limits.general]**

1. The headers `<limits>` (17.3.2), `<climits>` (17.3.5), and `<cfloat>` (17.3.6) supply characteristics of implementation-dependent arithmetic types (6.7.1).

2. The header `<version>` supplies implementation-dependent information about the C++ standard library (e.g., version number and release date).

3. The macros in Table 36 are defined after inclusion of the header `<version>` or one of the corresponding headers specified in the table. [Note: Future versions of this International Standard might replace the values of these macros with greater values. — end note]

#### Table 36 — Standard library feature-test macros

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_lib_addressof constexpr</td>
<td>201603L</td>
<td><code>&lt;memory&gt;</code>&lt;br&gt; <code>&lt;scoped_allocator&gt;</code>&lt;br&gt; <code>&lt;string&gt;</code>&lt;br&gt; <code>&lt;deque&gt;</code>&lt;br&gt; <code>&lt;forward_list&gt;</code>&lt;br&gt; <code>&lt;list&gt;</code>&lt;br&gt; <code>&lt;vector&gt;</code>&lt;br&gt; <code>&lt;map&gt;</code>&lt;br&gt; <code>&lt;set&gt;</code>&lt;br&gt; <code>&lt;unordered_map&gt;</code>&lt;br&gt; <code>&lt;unordered_set&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_allocator_traits_is_always_equal</td>
<td>201411L</td>
<td><code>&lt;memory&gt;</code>&lt;br&gt; <code>&lt;scoped_allocator&gt;</code>&lt;br&gt; <code>&lt;string&gt;</code>&lt;br&gt; <code>&lt;deque&gt;</code>&lt;br&gt; <code>&lt;forward_list&gt;</code>&lt;br&gt; <code>&lt;list&gt;</code>&lt;br&gt; <code>&lt;vector&gt;</code>&lt;br&gt; <code>&lt;map&gt;</code>&lt;br&gt; <code>&lt;set&gt;</code>&lt;br&gt; <code>&lt;unordered_map&gt;</code>&lt;br&gt; <code>&lt;unordered_set&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_array constexpr</td>
<td>201603L</td>
<td><code>&lt;iterator&gt;</code>&lt;br&gt; <code>&lt;array&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_as_const</td>
<td>201510L</td>
<td><code>&lt;utility&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_atomic_is_always_lock_free</td>
<td>201603L</td>
<td><code>&lt;atomic&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_atomic_ref</td>
<td>201806L</td>
<td><code>&lt;atomic&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_bit_cast</td>
<td>201806L</td>
<td><code>&lt;bit&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_bind_front</td>
<td>201811L</td>
<td><code>&lt;functional&gt;</code>&lt;br&gt;</td>
</tr>
<tr>
<td>__cpp_lib_bool_constant</td>
<td>201505L</td>
<td><code>&lt;type_traits&gt;</code>&lt;br&gt;</td>
</tr>
</tbody>
</table>

§ 17.3.1
Table 36 — Standard library feature-test macros (continued)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_lib_bounded_array_traits</td>
<td>201902L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_boyer_moore_searcher</td>
<td>201603L</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>__cpp_lib_byte</td>
<td>201603L</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>__cpp_lib_char8_t</td>
<td>201811L</td>
<td>&lt;atomic&gt; &lt;filesystem&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;istream&gt; &lt;limits&gt; &lt;locale&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;ostream&gt; &lt;string&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;string_view&gt;</td>
</tr>
<tr>
<td>__cpp_lib_chrono</td>
<td>201611L</td>
<td>&lt;chrono&gt;</td>
</tr>
<tr>
<td>__cpp_lib_chrono_udls</td>
<td>201304L</td>
<td>&lt;chrono&gt;</td>
</tr>
<tr>
<td>__cpp_lib_clamp</td>
<td>201603L</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>__cpp_lib_complex_udls</td>
<td>201309L</td>
<td>&lt;complex&gt;</td>
</tr>
<tr>
<td>__cpp_lib_concepts</td>
<td>201806L</td>
<td>&lt;concepts&gt;</td>
</tr>
<tr>
<td>__cpp_lib_constexpr_misc</td>
<td>201811L</td>
<td>&lt;array&gt; &lt;functional&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;iterator&gt; &lt;string_view&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;tuple&gt; &lt;utility&gt;</td>
</tr>
<tr>
<td>__cpp_lib_constexpr_swap_algorithms</td>
<td>201806L</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>__cpp_lib_destroying_delete</td>
<td>201806L</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>__cpp_lib_enable_shared_from_this</td>
<td>201603L</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>__cpp_lib_erase_if</td>
<td>201811L</td>
<td>&lt;string&gt; &lt;deque&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;forward_list&gt; &lt;list&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;vector&gt; &lt;map&gt; &lt;set&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;unordered_map&gt; &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>__cpp_lib_exchange_function</td>
<td>201304L</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>__cpp_lib_execution</td>
<td>201902L</td>
<td>&lt;execution&gt;</td>
</tr>
<tr>
<td>__cpp_lib_filesystem</td>
<td>201703L</td>
<td>&lt;filesystem&gt;</td>
</tr>
<tr>
<td>__cpp_lib_gcd_lcm</td>
<td>201606L</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>__cpp_lib_generic_associative_lookup</td>
<td>201304L</td>
<td>&lt;map&gt; &lt;set&gt;</td>
</tr>
<tr>
<td>__cpp_lib_generic_unordered_lookup</td>
<td>201811L</td>
<td>&lt;unordered_map&gt; &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>__cpp_lib_generic_unordered_hash_lookup</td>
<td>201902L</td>
<td>&lt;unordered_map&gt; &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>__cpp_lib_has_unique_object_representations</td>
<td>201606L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_hypot</td>
<td>201603L</td>
<td>&lt;cmath&gt;</td>
</tr>
<tr>
<td>__cpp_lib_incomplete_container_elements</td>
<td>201505L</td>
<td>&lt;forward_list&gt; &lt;list&gt; &lt;vector&gt;</td>
</tr>
<tr>
<td>__cpp_lib_integer_sequence</td>
<td>201304L</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>__cpp_lib_integral_constant_callable</td>
<td>201304L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_interpolate</td>
<td>201902L</td>
<td>&lt;cmath&gt; &lt;numeric&gt;</td>
</tr>
<tr>
<td>__cpp_lib_invoke</td>
<td>201411L</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>__cpp_lib_is_aggregate</td>
<td>201703L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_is_constant_evaluated</td>
<td>201811L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_is_final</td>
<td>201402L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_is_invocable</td>
<td>201703L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_is_null_pointer</td>
<td>201309L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_is_swappable</td>
<td>201603L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_launder</td>
<td>201606L</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>__cpp_lib_list_remove_return_type</td>
<td>201806L</td>
<td>&lt;forward_list&gt; &lt;list&gt;</td>
</tr>
<tr>
<td>__cpp_lib_logical_traits</td>
<td>201510L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_make_from_tuple</td>
<td>201606L</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>__cpp_lib_make_reverse_iterator</td>
<td>201402L</td>
<td>&lt;iterator&gt;</td>
</tr>
<tr>
<td>__cpp_lib_make_unique</td>
<td>201304L</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>__cpp_lib_map_try_emplace</td>
<td>201411L</td>
<td>&lt;map&gt;</td>
</tr>
</tbody>
</table>

§ 17.3.1
Table 36 — Standard library feature-test macros (continued)

<table>
<thead>
<tr>
<th>Macro name</th>
<th>Value</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__cpp_lib_math_special_functions</td>
<td>201603L</td>
<td>&lt;cmath&gt;</td>
</tr>
<tr>
<td>__cpp_lib_memory_resource</td>
<td>201603L</td>
<td>&lt;memory_resource&gt;</td>
</tr>
<tr>
<td>__cpp_lib_node_extract</td>
<td>201606L</td>
<td>&lt;map&gt; &lt;set&gt; &lt;unordered_map&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;unordered_set&gt;</td>
</tr>
<tr>
<td>__cpp_lib_nonmember_container_access</td>
<td>201411L</td>
<td>&lt;iterator&gt; &lt;array&gt; &lt;deque&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;forward_list&gt; &lt;list&gt; &lt;map&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;regex&gt; &lt;set&gt; &lt;string&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;unordered_map&gt; &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>__cpp_lib_not_fn</td>
<td>201603L</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>__cpp_lib_null_iterators</td>
<td>201304L</td>
<td>&lt;iterator&gt;</td>
</tr>
<tr>
<td>__cpp_lib_optional</td>
<td>201606L</td>
<td>&lt;optional&gt;</td>
</tr>
<tr>
<td>__cpp_lib_parallel_algorithm</td>
<td>201603L</td>
<td>&lt;algorithm&gt; &lt;numeric&gt;</td>
</tr>
<tr>
<td>__cpp_lib_quoted_string_io</td>
<td>201304L</td>
<td>&lt;iomanip&gt;</td>
</tr>
<tr>
<td>__cpp_lib_ranges</td>
<td>201811L</td>
<td>&lt;algorithm&gt; &lt;functional&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;iterator&gt; &lt;memory&gt; &lt;ranges&gt;</td>
</tr>
<tr>
<td>__cpp_lib_raw_memory_algorithms</td>
<td>201606L</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>__cpp_lib_result_of_sFINAE</td>
<td>201210L</td>
<td>&lt;functional&gt; &lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_robust_nonmodifying_seq_ops</td>
<td>201304L</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>__cpp_lib_sample</td>
<td>201603L</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>__cpp_lib_scoped_lock</td>
<td>201703L</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td>__cpp_lib_shared_mutex</td>
<td>201505L</td>
<td>&lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>__cpp_lib_shared_ptr_arrays</td>
<td>201611L</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>__cpp_lib_shared_ptr_weak_type</td>
<td>201606L</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>__cpp_lib_shared_timed_mutex</td>
<td>201402L</td>
<td>&lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>__cpp_lib_string_u8ds</td>
<td>201304L</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>__cpp_lib_string_view</td>
<td>201606L</td>
<td>&lt;string&gt; &lt;string_view&gt;</td>
</tr>
<tr>
<td>__cpp_lib_three_way_comparison</td>
<td>201711L</td>
<td>&lt;compare&gt;</td>
</tr>
<tr>
<td>__cpp_lib_to_chars</td>
<td>201611L</td>
<td>&lt;charconv&gt;</td>
</tr>
<tr>
<td>__cpp_lib_transformation_trait_aliases</td>
<td>201304L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_transparent_operators</td>
<td>201510L</td>
<td>&lt;memory&gt; &lt;functional&gt;</td>
</tr>
<tr>
<td>__cpp_lib_tuple_element_t</td>
<td>201402L</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>__cpp_lib_tuples_by_type</td>
<td>201304L</td>
<td>&lt;utility&gt; &lt;tuple&gt;</td>
</tr>
<tr>
<td>__cpp_lib_type_trait_variable_templates</td>
<td>201510L</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>__cpp_lib_uncaught_exceptions</td>
<td>201411L</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td>__cpp_lib_unordered_map_try_emplace</td>
<td>201411L</td>
<td>&lt;unordered_map&gt;</td>
</tr>
<tr>
<td>__cpp_lib_variant</td>
<td>201606L</td>
<td>&lt;variant&gt;</td>
</tr>
<tr>
<td>__cpp_lib_void_t</td>
<td>201411L</td>
<td>&lt;type_traits&gt;</td>
</tr>
</tbody>
</table>

17.3.2 Header <limits> synopsis

```cpp
namespace std {
    // 17.3.3, floating-point type properties
    enum float_round_style;
    enum float_denorm_style;

    // 17.3.4, class template numeric_limits
template<class T> class numeric_limits;
    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>;
}
```

§ 17.3.2
template<> class numeric_limits<char16_t>;
template<> class numeric_limits<char32_t>;
template<> class numeric_limits<wchar_t>;
template<> class numeric_limits<short>;
template<> class numeric_limits<int>;
template<> class numeric_limits<long>;
template<> class numeric_limits<long long>;
template<> class numeric_limits<unsigned short>;
template<> class numeric_limits<unsigned int>;
template<> class numeric_limits<unsigned long>;
template<> class numeric_limits<unsigned long long>;

template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;

17.3.3 Floating-point type properties [fp.style]

17.3.3.1 Type float_round_style [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

17.3.3.2 Type float_denorm_style [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

17.3.4 Class template numeric_limits [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;
};

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>;

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.

17.3.4.1 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.188

3 For floating-point types with subnormal numbers, returns the minimum positive normalized value.

188) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
Meaningful for all specializations in which \( \text{is\_bounded} \neq \text{false} \), or \( \text{is\_bounded} = \text{false} \text{ and } \text{is\_signed} = \text{false} \).

```
static constexpr T max() noexcept;
```

Maximum finite value.\(^{189}\)

Meaningful for all specializations in which \( \text{is\_bounded} \neq \text{false} \).

```
static constexpr T lowest() noexcept;
```

A finite value \( x \) such that there is no other finite value \( y \) where \( y < x \).\(^{190}\)

Meaningful for all specializations in which \( \text{is\_bounded} \neq \text{false} \).

```
static constexpr int digits;
```

Number of \( \text{radix} \) 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 \( \text{radix} \) digits in the mantissa.\(^{191}\)

```
static constexpr int digits10;
```

Number of base 10 digits that can be represented without change.\(^{192}\)
Meaningful for all specializations in which \( \text{is\_bounded} \neq \text{false} \).

```
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).\(^{193}\)
For integer types, specifies the base of the representation.\(^{194}\)
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.\(^{195}\)
Meaningful for all floating-point types.

---

\(^{189}\) Equivalent to \texttt{CHAR\_MAX}, \texttt{SHRT\_MAX}, \texttt{FLT\_MAX}, \texttt{DBL\_MAX}, etc.

\(^{190}\) \texttt{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.

\(^{191}\) Equivalent to \texttt{FLT\_MANT\_DIG}, \texttt{DBL\_MANT\_DIG}, \texttt{LDBL\_MANT\_DIG}.

\(^{192}\) Equivalent to \texttt{FLT\_DIG}, \texttt{DBL\_DIG}, \texttt{LDBL\_DIG}.

\(^{193}\) Equivalent to \texttt{FLT\_RADIX}.

\(^{194}\) Distinguishes types with bases other than 2 (e.g. BCD).

\(^{195}\) Equivalent to \texttt{FLT\_EPSILON}, \texttt{DBL\_EPSILON}, \texttt{LDBL\_EPSILON}.
static constexpr T round_error() noexcept;

Measure of the maximum rounding error.\(^{196}\)

static constexpr int min_exponent;

Minimum negative integer such that \texttt{radix} raised to the power of one less than that integer is a
normalized floating-point number.\(^{197}\)

Meaningful for all floating-point types.

static constexpr int min_exponent10;

Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point
numbers.\(^{198}\)

Meaningful for all floating-point types.

static constexpr int max_exponent;

Maximum positive integer such that \texttt{radix} raised to the power one less than that integer is a representable
finite floating-point number.\(^{199}\)

Meaningful for all floating-point types.

static constexpr int max_exponent10;

Maximum positive integer such that 10 raised to that power is in the range of representable finite
floating-point numbers.\(^{200}\)

Meaningful for all floating-point types.

static constexpr bool has_infinity;

true if the type has a representation for positive infinity.

Meaningful for all floating-point types.

Shall be true for all specializations in which \texttt{is_iec559} != false.

static constexpr bool has_quiet_NaN;

true if the type has a representation for a quiet (non-signaling) “Not a Number”.\(^{201}\)

Meaningful for all floating-point types.

Shall be true for all specializations in which \texttt{is_iec559} != false.

static constexpr bool has_signaling_NaN;

true if the type has a representation for a signaling “Not a Number”.\(^{202}\)

Meaningful for all floating-point types.

Shall be true for all specializations in which \texttt{is_iec559} != false.

static constexpr float_denorm_style has_denorm;

denorm_present if the type allows subnormal values (variable number of exponent bits)\(^{203}\),
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.

static constexpr bool has_denorm_loss;

true if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.\(^{204}\)

\(^{196}\) Rounding error is described in LIA-1 Section 5.2.4 and Annex C Rationale Section C.5.2.4 — Rounding and rounding
constants.

\(^{197}\) Equivalent to \texttt{FLT_MIN_EXP}, \texttt{DBL_MIN_EXP}, \texttt{LDBL_MIN_EXP}.

\(^{198}\) Equivalent to \texttt{FLT_MIN_10_EXP}, \texttt{DBL_MIN_10_EXP}, \texttt{LDBL_MIN_10_EXP}.

\(^{199}\) Equivalent to \texttt{FLT_MAX_EXP}, \texttt{DBL_MAX_EXP}, \texttt{LDBL_MAX_EXP}.

\(^{200}\) Equivalent to \texttt{FLT_MAX_10_EXP}, \texttt{DBL_MAX_10_EXP}, \texttt{LDBL_MAX_10_EXP}.

\(^{201}\) Required by LIA-1.

\(^{202}\) Required by LIA-1.

\(^{203}\) Required by LIA-1.

\(^{204}\) See ISO/IEC/IEEE 60559.
static constexpr T infinity() noexcept;
    Representation of positive infinity, if available.\(^{205}\)
    Meaningful for all specializations for which has_infinity != false. Required in specializations for which is_iec559 != false.

static constexpr T quiet_NaN() noexcept;
    Representation of a quiet “Not a Number”, if available.\(^{206}\)
    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.\(^{207}\)
    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.\(^{208}\)
    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.\(^{209}\)
    Meaningful for all floating-point types.

static constexpr bool is_bounded;
    true if the set of values representable by the type is finite.\(^{210}\) [Note: All fundamental types (6.7.1) 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.\(^{211}\) A type is modulo if, for any operation involving +, -, or * on values of that type whose result would fall outside the range \([\text{min}()\), \text{max}()]\), the value returned differs from the true value by an integer multiple of \(\text{max}() - \text{min}() + 1\).

    [Example: is_modulo is false for signed integer types (6.7.1) 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 program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.\(^{212}\)
    Meaningful for all specializations.

static constexpr bool tinyness_before;
    true if tinyness is detected before rounding.\(^{213}\)
    Meaningful for all floating-point types.

\(^{205}\) Required by LIA-1.
\(^{206}\) Required by LIA-1.
\(^{207}\) Required by LIA-1.
\(^{208}\) Required by LIA-1.
\(^{210}\) Required by LIA-1.
\(^{211}\) Required by LIA-1.
\(^{212}\) Required by LIA-1.
\(^{213}\) Refer to ISO/IEC/IEEE 60559. Required by LIA-1.
static constexpr float_round_style round_style;

The rounding style for the type.\footnote{Equivalent to FLT_ROUNDS. Required by LIA-1.}
Meaningful for all floating-point types. Specializations for integer types shall return \texttt{round\_toward\_zero}.

17.3.4.2 \texttt{numeric\_limits} specializations

All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, \texttt{epsilon()} is only meaningful if \texttt{is\_integer} is \texttt{false}). Any value that is not "meaningful" shall be set to 0 or \texttt{false}.

\texttt{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_infinity = true;
            static constexpr bool has_quiet_NaN = true;
            static constexpr bool has_signaling_NaN = true;

            static constexpr float_denorm_style has_denorm = denorm_absent;
            static constexpr bool has_denorm_loss = false;

            static constexpr float infinity() noexcept { return \texttt{value}; }
            static constexpr float quiet_NaN() noexcept { return \texttt{value}; }
            static constexpr float signaling_NaN() noexcept { return \texttt{value}; }
            static constexpr float denorm_min() noexcept { return \texttt{min}; }

            static constexpr bool is_iec559 = true;
            static constexpr bool is_bounded = true;
            static constexpr bool traps = true;
            static constexpr bool tinyness_before = true;

            static constexpr float_round_style round_style = round_to_nearest;
    };
}
```

\texttt{Example:}

The specialization for \texttt{bool} shall be provided as follows:
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; }
            static constexpr int digits = 1;
            static constexpr int digits10 = 0;
            static constexpr int max_digits10 = 0;
            static constexpr bool is_signed = false;
            static constexpr bool is_integer = true;
            static constexpr bool is_exact = true;
            static constexpr int radix = 2;
            static constexpr bool epsilon() noexcept { return 0; }
            static constexpr bool round_error() noexcept { return 0; }
            static constexpr int min_exponent = 0;
            static constexpr int min_exponent10 = 0;
            static constexpr int max_exponent = 0;
            static constexpr int max_exponent10 = 0;
            static constexpr bool has_infinity = false;
            static constexpr bool has_quiet_NaN = false;
            static constexpr bool has_signaling_NaN = false;
            static constexpr float_denorm_style has_denorm = denorm_absent;
            static constexpr bool has_denorm_loss = false;
            static constexpr bool infinity() noexcept { return 0; }
            static constexpr bool quiet_NaN() noexcept { return 0; }
            static constexpr bool signaling_NaN() noexcept { return 0; }
            static constexpr bool denorm_min() noexcept { return 0; }
            static constexpr bool is_iec559 = false;
            static constexpr bool is_bounded = true;
            static constexpr bool is_modulo = false;
            static constexpr bool traps = false;
            static constexpr float_round_style round_style = round_toward_zero;
    }
}

17.3.5 Header <climits> synopsis

#define CHAR_BIT see below
#define SCHAR_MIN see below
#define SCHAR_MAX see below
#define UCHAR_MAX see below
#define CHAR_MIN see below
#define CHAR_MAX see below
#define MB_LEN_MAX see below
#define SHRT_MIN see below
#define SHRT_MAX see below
#define USHRT_MAX see below
#define INT_MIN see below
#define INT_MAX see below
#define UINT_MAX see below
#define LONG_MIN see below
#define LONG_MAX see below
#define ULONG_MAX see below
#define LLONG_MIN see below
#define LLONG_MAX see below
#define ULLONG_MAX see below
1 The header `<climits>` defines all macros the same as the C standard library header `<limits.h>`. [Note: 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

17.3.6 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 FLT_DIG see below
#define DBL_DIG see below
#define LDBL_DIG see below
#define FLT_MIN_EXP see below
#define DBL_MIN_EXP see below
#define LDBL_MIN_EXP see below
#define FLT_MIN_10_EXP see below
#define DBL_MIN_10_EXP see below
#define LDBL_MIN_10_EXP see below
#define FLT_MAX_EXP see below
#define DBL_MAX_EXP see below
#define LDBL_MAX_EXP see below
#define FLT_MAX see below
#define DBL_MAX see below
#define LDBL_MAX see below
#define FLT_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

```cpp
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;
}
```
The header also defines numerous macros of the form:

```
INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT(MAX PTR)_MIN
[U]INT(MAX PTR)_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX
```

plus function macros of the form:

```
[U]INT{8 16 32 64 MAX}_C
```

The header defines all types and macros the same as the C standard library header `<stdint.h>`.

**17.5 Start and termination**

[**support.start.term**]

1. The header also defines numerous macros of the form:

   ```
   INT_[FAST LEAST]{8 16 32 64}_MIN
   [U]INT_[FAST LEAST]{8 16 32 64}_MAX
   INT(MAX PTR)_MIN
   [U]INT(MAX PTR)_MAX
   {PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
   SIZE_MAX
   ```

   plus function macros of the form:

   ```
   [U]INT{8 16 32 64 MAX}_C
   ```

2. The header defines all types and macros the same as the C standard library header `<stdint.h>`.

**See also:** ISO C 7.20

### 17.5.1 Start and termination

1. **Note:** The header `<cstdlib>` (17.2.2) declares the functions described in this subclause. — end note]

   ```
   [[noreturn]] void _Exit(int status) noexcept;
   ```

   **Effects:** This function has the semantics specified in the C standard library.

   **Remarks:** The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (6.8.3.4). The function `_Exit` is signal-safe (17.13.4).

   ```
   [[noreturn]] void abort() noexcept;
   ```

   **Effects:** This function has the semantics specified in the C standard library.

   **Remarks:** The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling functions passed to `atexit()` (6.8.3.4). The function `abort` is signal-safe (17.13.4).

2. ```
   int atexit(c-atexit-handler* f) noexcept;
   int atexit(atexit-handler* f) noexcept;
   ```

   **Effects:** The `atexit()` functions register the function pointed to by `f` to be called without arguments at normal program termination. It is unspecified whether a call to `atexit()` that does not happen
before (6.8.2) a call to exit() will succeed. [Note: The atexit() functions do not introduce a data race (16.5.5.9). — end note]

Implementation limits: The implementation shall support the registration of at least 32 functions.

Returns: The atexit() function returns zero if the registration succeeds, nonzero if it fails.

[[noreturn]] void exit(int status);

Effects:

(9.1) — First, objects with thread storage duration and associated with the current thread are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling atexit are called. See 6.8.3.4 for the order of destructions and calls. (Automatic objects 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 (14.5.1).

(9.2) — Next, all open C streams (as mediated by the function signatures declared in <cstdio>) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling tmpfile() are removed.

(9.3) — Finally, control is returned to the host environment. If status is zero or EXIT_SUCCESS, an implementation-defined form of the status successful termination is returned. If status is EXIT_FAILURE, an implementation-defined form of the status unsuccessful termination is returned. Otherwise the status returned is implementation-defined.

int at_quick_exit(c-atexit-handler* f) noexcept;
int at_quick_exit(atexit-handler* f) noexcept;

Effects: The at_quick_exit() functions register the function pointed to by f to be called without arguments when quick_exit is called. It is unspecified whether a call to at_quick_exit() that does not happen before (6.8.2) all calls to quick_exit will succeed. [Note: The at_quick_exit() functions do not introduce a data race (16.5.5.9). — 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.

[[noreturn]] void quick_exit(int status) noexcept;

Effects: Functions registered by calls to at_quick_exit are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling quick_exit. If control leaves a registered function called by quick_exit because the function does not provide a handler for a thrown exception, the function std::terminate shall be called. [Note: 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

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.

215) A function is called for every time it is registered.

216) Objects with automatic storage duration are all destroyed in a program whose main function (6.8.3.1) contains no automatic objects 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.

217) The macros EXIT_FAILURE and EXIT_SUCCESS are defined in <cstdlib>.
17.6.1 Header `<new>` synopsis

```cpp
namespace std {
    class bad_alloc;
    class bad_array_new_length;

    struct destroying_delete_t {
        explicit destroying_delete_t() = default;
    };
    inline constexpr destroying_delete_t destroying_delete{};

    enum class align_val_t : size_t {};
    struct nothrow_t { explicit nothrow_t() = default; };
    extern 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;
}
```

```cpp
[[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;
```
17.6.2.1 Single-object forms

[[nodiscard]] void* operator new(std::size_t size);
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment);

Effects: The allocation functions (6.6.5.4.1) called by a new-expression (7.6.2.5) to allocate size bytes of storage. The second form is called for a type with new-extended alignment, and allocates storage with the specified alignment. The first form is called otherwise, and allocates storage suitably aligned to represent any object of that size provided the object’s type does not have new-extended alignment.

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 (6.6.5.4), or else throw a bad_alloc exception. This requirement is binding on any replacement versions of these functions.

Default behavior:

(4.1) Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the C standard library functions malloc or aligned_alloc is unspecified.

(4.2) Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the current new_handler (17.6.3.5) is a null pointer value, throws bad_alloc.

(4.3) Otherwise, the function calls the current new_handler function (17.6.3.3). If the called function returns, the loop repeats.

(4.4) The loop terminates when an attempt to allocate the requested storage is successful or when a called new_handler function does not return.

[[nodiscard]] void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;

Effects: Same as above, except that these are called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad_alloc exception.

Replaceable: A C++ program may define functions with any 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 (6.6.5.4), 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.

Example:

```cpp
T* p1 = new T; // throws bad_alloc if it fails
T* p2 = new(nothrow) T; // returns nullptr if it fails
```

§ 17.6.2.1 491
Default 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. [Note: See the note in the above Replaceable: paragraph. — end note]

Default behavior: If ptr is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to operator new.

Remarks: It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to operator new or any of aligned_alloc, calloc, malloc, or realloc, declared in <cstdlib>.

### 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.

Expects: 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.

Expects: If an implementation has strict pointer safety (6.6.5.4.3) then ptr is a safely-derived pointer.

Expects: 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.

### Replaceable

Expects: 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.

Expects: If an implementation has strict pointer safety (6.6.5.4.3) then ptr is a safely-derived pointer.

Expects: 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.

Default behavior: Calls operator delete(ptr), or operator delete(ptr, alignment), respectively.

### 17.6.2.2 Array forms

[[nodiscard]] void* operator new[](std::size_t size);
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment);

Effects: The allocation functions (6.6.5.4.1) called by the array form of a new-expression (7.6.2.5) to allocate size bytes of storage. The second form is called for a type with new-extended alignment, and allocates storage with the specified alignment. The first form is called otherwise, and allocates storage suitably aligned to represent any array object of that size or smaller, provided the object’s type does not have new-extended alignment.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Required behavior: Same as for the corresponding single-object forms. This requirement is binding on any replacement versions of these functions.

Default behavior: Returns operator new(size), or operator new(size, alignment), respectively.

---

218 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.

§ 17.6.2.2
[[nodiscard]] void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment, const std::nothrow_t&) noexcept;

Effects: Same as above, except that these are called by a placement version of a new-expression when a C++ program prefers a null pointer result as an error indication, instead of a bad_alloc exception.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Required behavior: Return a non-null pointer to suitably aligned storage (6.6.5.4), 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 (6.6.5.4.2) 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]

Expects: 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[].

Effects: If an implementation has strict pointer safety (6.6.5.4.3) then ptr is a safely-derived pointer.

Expects: 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 (6.6.5.4.2) 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.

Expects: 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[].

Expects: If an implementation has strict pointer safety (6.6.5.4.3) then ptr is a safely-derived pointer.
Expects: 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`.

Default behavior: Calls `operator delete[](ptr)`, or `operator delete[](ptr, 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 6.6.5.4 do not apply to these reserved placement forms of `operator new` and `operator delete`.

```cpp
[[nodiscard]] void* operator new(std::size_t size, void* ptr) noexcept;
Returns: `ptr`.
Remarks: Intentionally performs no other action.
[Example: This can be useful for constructing an object at a known address:
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
—end example]
[[nodiscard]] void* operator new[](std::size_t size, void* ptr) noexcept;
Returns: `ptr`.
Remarks: Intentionally performs no other action.
```
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.

Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a `wstring` (21.3, 28.4.1.4).

### 17.6.3.2 Class `bad_array_new_length`

```cpp
namespace std {

    class bad_array_new_length : public bad_alloc {
        public:
            bad_array_new_length() noexcept;
            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 (7.6.2.5).

```cpp
const char* what() const noexcept override;
```

Returns: An implementation-defined NTBS.

Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a `wstring` (21.3, 28.4.1.4).

### 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:

1. make more storage available for allocation and then return;
2. throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;
3. 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.

Expects: `p` represents the address `A` of a byte in memory. An object `X` that is within its lifetime (6.6.3) 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`.

§ 17.6.4
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 is reachable through a pointer value that points to an object \( Y \) if it is within the storage occupied by \( Y \), an object that is pointer-interconvertible with \( Y \), or the immediately-enclosing array object if \( Y \) 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 the type contains const or reference members; in the latter cases, this function can be used to obtain a usable pointer to the new object. See 6.6.3. — end note]

[Example:
```c++
struct X { const int n; };
X *p = new X{3};
const int a = p->n;
new (p) X{5}; // p does not point to new object (6.6.3) because X::n is const
const int b = p->n; // undefined behavior
const int c = std::launder(p)->n; // OK
```
—end example]

17.6.5 Hardware interference size

inline constexpr size_t hardware_destructive_interference_size = implementation-defined;

This number is the minimum recommended offset between two concurrently-accessed objects to avoid additional performance degradation due to contention introduced by the implementation. It shall be at least `alignof(max_align_t)`.

[Example:
```c++
struct keep_apart {
    alignas(hardware_destructive_interference_size) atomic<int> cat;
    alignas(hardware_destructive_interference_size) atomic<int> dog;
};
```
—end example]

inline constexpr size_t hardware_constructive_interference_size = implementation-defined;

This number is the maximum recommended size of contiguous memory occupied by two objects accessed with temporal locality by concurrent threads. It shall be at least `alignof(max_align_t)`.

[Example:
```c++
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

```c++
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```
17.7.2 Class type_info

namespace std {
    class type_info {
    public:
        virtual ~type_info();
        bool operator==(const type_info& rhs) const noexcept;
        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& rhs) = delete; // cannot be copied
        type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    }; // type_info

    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;
    Effects: Compares the current object with rhs.
    Returns: true if the two values describe the same type.

    bool operator!=(const type_info& rhs) const noexcept;
    Returns: !(this == rhs).

    bool before(const type_info& rhs) const noexcept;
    Effects: Compares the current object with rhs.
    Returns: true if *this precedes rhs in the implementation’s collation order.

    size_t hash_code() const noexcept;
    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.

    const char* name() const noexcept;
    Returns: An implementation-defined NTBS.

    Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a wstring (21.3, 28.4.1.4)

17.7.3 Class bad_cast

namespace std {
    class bad_cast : public exception {
    public:
        bad_cast() noexcept;
        bad_cast(const bad_cast&) noexcept;
        bad_cast& operator=(const bad_cast&) noexcept;
        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;
    Returns: An implementation-defined NTBS.
17.7.4 Class bad_typeid

namespace std {
  class bad_typeid : public exception {
public:
  bad_typeid() noexcept;
  bad_typeid(const bad_typeid&) noexcept;
  bad_typeid& operator=(const bad_typeid&) noexcept;
  const char* what() const noexcept override;
  }
}

The class bad_typeid defines the type of objects thrown as exceptions by the implementation to report a null pointer in a typeid expression (7.6.1.7).

const char* what() const noexcept override;

Returns: An implementation-defined ntbs.

Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a wstring (21.3, 28.4.1.4)

17.8 Contract violation handling

17.8.1 Header <contract> synopsis

The header <contract> defines a type for reporting information about contract violations generated by the implementation.

namespace std {
  class contract_violation;
}

17.8.2 Class contract_violation

namespace std {
  class contract_violation {
public:
  uint_least32_t line_number() const noexcept;
  string_view file_name() const noexcept;
  string_view function_name() const noexcept;
  string_view comment() const noexcept;
  string_view assertion_level() const noexcept;
  }
}

The class contract_violation describes information about a contract violation generated by the implementation.

uint_least32_t line_number() const noexcept;

Returns: The source code location where the contract violation happened (9.11.4). If the location is unknown, an implementation may return 0.

string_view file_name() const noexcept;

Returns: The source file name where the contract violation happened (9.11.4). If the file name is unknown, an implementation may return string_view{}.

string_view function_name() const noexcept;

Returns: The name of the function where the contract violation happened (9.11.4). If the function name is unknown, an implementation may return string_view{}.

string_view comment() const noexcept;

Returns: Implementation-defined text describing the predicate of the violated contract.
string_view assertion_level() const noexcept;

Returns: Text describing the assertion-level of the violated contract.

17.9 Exception handling [support.exception]

The header <exception> defines several types and functions related to the handling of exceptions in a C++ program.

17.9.1 Header <exception> synopsis [exception.syn]

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 [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;
  };
}

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.

Each standard library class T that derives from class exception shall have a publicly accessible copy constructor and a publicly accessible copy assignment operator that do not exit with an exception. These member functions shall meet the following postcondition: If two objects lhs and rhs both have dynamic type T and lhs is a copy of rhs, then strcmp(lhs.what(), rhs.what()) shall equal 0.

exception(const exception& rhs) noexcept;
exception& operator=(const exception& rhs) noexcept;

Ensures: If *this and rhs both have dynamic type exception then the value of the expression strcmp(what(), rhs.what()) shall equal 0.

virtual ~exception();

Effects: Destroys an object of class exception.

virtual const char* what() const noexcept;

Returns: An implementation-defined NTBS.
Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a 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-const member function of the exception object is called.

17.9.3 Class bad_exception

namespace std {
    class bad_exception : public exception {
    public:
        bad_exception() noexcept;
        bad_exception(const bad_exception&) noexcept;
        bad_exception& operator=(const bad_exception&) noexcept;
        const char* what() const noexcept override;
    };
}

The class bad_exception defines the type of the object referenced by the exception_ptr returned from a call to current_exception (17.9.6) when the currently active exception object fails to copy.

const char* what() const noexcept override;

Returns: An implementation-defined nTBS.

Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a wstring (21.3, 28.4.1.4).

17.9.4 Abnormal termination

17.9.4.1 Type terminate_handler

using terminate_handler = void (*)(void);

The type of a handler function to be called by std::terminate() when terminating exception processing.

Required behavior: A terminate_handler shall terminate execution of the program without returning to the caller.

Default behavior: The implementation’s default terminate_handler calls abort().

17.9.4.2 set_terminate

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 terminate_handler.

Returns: The previous terminate_handler.

17.9.4.3 get_terminate

terminate_handler get_terminate() noexcept;

Returns: The current terminate_handler. [Note: This may be a null pointer value. — end note]

17.9.4.4 terminate

[[noreturn]] void terminate() noexcept;

Remarks: Called by the implementation when exception handling must be abandoned for any of several reasons (14.5.1). May also be called directly by the program.

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

```cpp
int uncaught_exceptions() noexcept;
```

*Returns:* The number of uncaught exceptions (14.5.2).

*Remarks:* When uncaught_exceptions() > 0, throwing an exception can result in a call of the function std::terminate (14.5.1).

17.9.6 Exception propagation

```cpp
using exception_ptr = unspecified;
```

The type exception_ptr can be used to refer to an exception object.

```cpp
exception_ptr meets the requirements of Cpp17NullablePointer (Table 31).
```

Two non-null values of type exception_ptr are equivalent and compare equal if and only if they refer to the same exception.

The default constructor of exception_ptr produces the null value of the type.

```cpp
exception_ptr shall not be implicitly convertible to any arithmetic, enumeration, or pointer type.
```

*Note:* An implementation might use a reference-counted smart pointer as exception_ptr. — end note

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]

```cpp
exception_ptr current_exception() noexcept;
```

*Returns:* An exception_ptr object that refers to the currently handled exception (14.3) 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]

```cpp
[noreturn] void rethrow_exception(exception_ptr p);
```

*Expects:* p is not a null pointer.

*Throws:* The exception object to which p refers.

```cpp
template<class E> exception_ptr make_exception_ptr(E e) noexcept;
```

*Effects:* Creates an exception_ptr object that refers to a copy of e, as if:

```cpp
try {
    throw e;
} catch(...) {
    return current_exception();
}
```

*Note:* This function is provided for convenience and efficiency reasons. — end note

17.9.7 nested_exception

```cpp
namespace std {
    class nested_exception {
        public:
```
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

### Effects: The constructor calls `current_exception()` and stores the returned value.

`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`.

### Returns: The stored exception captured by this `nested_exception` object.

`exception_ptr nested_ptr()` const noexcept;

### template<class T> void throw_with_nested(T&& t);

**Let:** `U` be `decay_t<T>`.

**Expects:** `U` meets the `Cpp17CopyConstructible` requirements.

**Throws:** If `is_class_v<U> && !is_final_v<U> && !is_base_of_v<nested_exception, U>` is true, an exception of unspecified type that is publicly derived from both `U` and `nested_exception` and constructed from `std::forward<T>(t)`, otherwise `std::forward<T>(t)`.

`void rethrow_if_nested(const E& e);`

**Effects:** If `E` is not a polymorphic class type, or if `nested_exception` is an inaccessible or ambiguous base class of `E`, there is no effect. Otherwise, performs:

```
if (auto p = dynamic_cast<const nested_exception*>(addressof(e)))
    p->rethrow_nested();
```

### Initializer lists

The header `<initializer_list>` defines a class template and several support functions related to list-initialization (see 9.3.4). All functions specified in this subclause are signal-safe (17.13.4).

### Header `<initializer_list>` synopsis

```cpp
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;

            template<class T> [[noreturn]] void throw_with_nested(T&& t);
            template<class E> void rethrow_if_nested(const E& e);
    }
}
```
```cpp
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;
}

1 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 9.3.4. Copying an initializer list does not copy the underlying elements. — end note]

2 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;

1 Ensures: size() == 0.

17.10.3 Initializer list access
constexpr const E* begin() const noexcept;

1 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;

2 Returns: begin() + size().

constexpr size_t size() const noexcept;

3 Returns: The number of elements in the array.
4 Complexity: Constant time.

17.10.4 Initializer list range access
template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;

1 Returns: il.begin().

template<class E> constexpr const E* end(initializer_list<E> il) noexcept;

2 Returns: il.end().

17.11 Comparisons

17.11.1 Header <compare> synopsis

1 The header <compare> specifies types, objects, and functions for use primarily in connection with the three-way comparison operator (7.6.8).

namespace std {
    // 17.11.2, comparison category types
    class weak_equality;
    class strong_equality;
    class partial_ordering;
    class weak_ordering;
    class strong_ordering;

    // named comparison functions
    constexpr bool is_eq (weak_equality cmp) noexcept { return cmp == 0; }
    constexpr bool is_neq (weak_equality 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, comparison algorithms
template<class T> constexpr strong_ordering strong_order(const T& a, const T& b);
template<class T> constexpr weak_ordering weak_order(const T& a, const T& b);
template<class T> constexpr partial_ordering partial_order(const T& a, const T& b);
template<class T> constexpr strong_equality strong_equal(const T& a, const T& b);
template<class T> constexpr weak_equality weak_equal(const T& a, const T& b);

17.11.2 Comparison category types

17.11.2.1 Preamble

The types weak_equality, strong_equality, 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 types strong_ordering and weak_equality correspond, respectively, to the terms total ordering
and equivalence 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 satisfies 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 this subclause, 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 weak_equality

The weak_equality type is typically used as the result type of a three-way comparison operator (7.6.8) that
(a) admits only equality and inequality comparisons, and (b) does not imply substitutability.

namespace std {
    class weak_equality {
        int value; // exposition only
        // exposition-only constructor
        constexpr explicit weak_equality(eq v) noexcept : value(int(v)) {} // exposition only
    public:
        // valid values
        static const weak_equality equivalent;
        static const weak_equality nonequivalent;
        // comparisons
        friend constexpr explicit operator==(weak_equality v, unspecified) noexcept;
        friend constexpr explicit operator!=(weak_equality v, unspecified) noexcept;
        friend constexpr explicit operator==(unspecified, weak_equality v) noexcept;
        friend constexpr explicit operator!=(unspecified, weak_equality v) noexcept;
    }
}
friend constexpr weak_equality operator<=>(weak_equality v, unspecified) noexcept;
friend constexpr weak_equality operator<=>(unspecified, weak_equality v) noexcept;

// valid values' definitions
inline constexpr weak_equality weak_equality::equivalent(eq::equivalent);
inline constexpr weak_equality weak_equality::nonequivalent(eq::nonequivalent);
}

constexpr bool operator==(weak_equality v, unspecified) noexcept;
constexpr bool operator==(unspecified, weak_equality v) noexcept;

Returns: v.value == 0.

constexpr bool operator!=(weak_equality v, unspecified) noexcept;
constexpr bool operator!=(unspecified, weak_equality v) noexcept;

Returns: v.value != 0.

constexpr weak_equality operator<=>(weak_equality v, unspecified) noexcept;
constexpr weak_equality operator<=>(unspecified, weak_equality v) noexcept;

Returns: v.

17.11.2.3 Class strong_equality

The strong_equality type is typically used as the result type of a three-way comparison operator (7.6.8) that (a) admits only equality and inequality comparisons, and (b) does imply substitutability.

namespace std {
    class strong_equality {
        int value; // exposition only

        // exposition-only constructor
        constexpr explicit strong_equality(eq v) noexcept : value(int(v)) {}
        // exposition only
    public:
        // valid values
        static const strong_equality equal;
        static const strong_equality nonequal;
        static const strong_equality equivalent;
        static const strong_equality nonequivalent;
        // conversion
        constexpr operator weak_equality() const noexcept;
        // comparisons
        friend constexpr bool operator==(strong_equality v, unspecified) noexcept;
        friend constexpr bool operator==(unspecified, strong_equality v) noexcept;
    
    // valid values' definitions
    inline constexpr strong_equality strong_equality::equal(eq::equal);
    inline constexpr strong_equality strong_equality::nonequal(eq::nonequal);
    inline constexpr strong_equality strong_equality::equivalent(eq::equivalent);
    inline constexpr strong_equality strong_equality::nonequivalent(eq::nonequivalent);
    }

conestexpr operator weak_equality() const noexcept;

    Returns: value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent.

    constexpr bool operator==(strong_equality v, unspecified) noexcept;

§ 17.11.2.3
17.11.2.4 Class partial_ordering

The \texttt{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.\footnote{That is, \texttt{a < b}, \texttt{a == b}, and \texttt{a > b} might all be false.}

```cpp
namespace std {
    class partial_ordering {
        // valid values
        static const partial_ordering less;
        static const partial_ordering equivalent;
        static const partial_ordering greater;
        static const partial_ordering unordered;

        // conversion
        constexpr operator weak_equality() const noexcept;

        // 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>=(partial_ordering v, unspecified) noexcept;

        // valid values' definitions
        inline constexpr partial_ordering partial_ordering::less(ord::less);
        inline constexpr partial_ordering partial_ordering::equivalent(eq::equivalent);
        inline constexpr partial_ordering partial_ordering::greater(ord::greater);
    };
}
```
inline constexpr partial_ordering partial_ordering::unordered(ncmp::unordered);
}

constexpr operator weak_equality() const noexcept;

Returns: value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent. [Note: The result is independent of the is_ordered member. — end note]

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;

Returns: For operator@, v.is_ordered && v.value @ 0.

constexpr bool operator==(unspecified, partial_ordering v) noexcept;
constexpr bool operator<(unspecified, partial_ordering v) noexcept;
constexpr bool operator>(unspecified, partial_ordering v) noexcept;
constexpr bool operator<=(unspecified, partial_ordering v) noexcept;
constexpr bool operator>=(unspecified, partial_ordering v) noexcept;

Returns: For operator@, !v.is_ordered || v.value != 0.

constexpr partial_ordering operator<>=(partial_ordering v, unspecified) noexcept;

Returns: v < 0 ? partial_ordering::greater : v > 0 ? partial_ordering::less : v.

17.11.2.5 Class weak_ordering

The weak_ordering type is typically used as the result type of a three-way comparison operator (7.6.8) that (a) admits all of the six two-way comparison operators (7.6.9, 7.6.10), and (b) does not imply substitutability.

namespace std {
    class weak_ordering {
        int value; // exposition only

        // exposition-only constructors
        constexpr explicit weak_ordering(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 weak_equality() const noexcept;
        constexpr operator partial_ordering() const noexcept;

        // comparisons
        friend constexpr bool operator==(weak_ordering v, unspecified) noexcept;
        friend constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
        friend constexpr bool operator<(weak_ordering v, unspecified) noexcept;
        friend constexpr bool operator>(weak_ordering v, unspecified) noexcept;
        friend constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
        friend constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
        friend constexpr bool operator==(unspecified, weak_ordering v) noexcept;
        friend constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
    }
friend constexpr bool operator< (unspecified, weak_ordering v) noexcept;
friend constexpr bool operator> (unspecified, weak_ordering v) noexcept;
friend constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
friend constexpr bool operator>=(unspecified, weak_ordering v) noexcept;
friend constexpr weak_ordering operator<=>(weak_ordering v, unspecified) noexcept;
friend constexpr weak_ordering operator<=>(unspecified, weak_ordering v) noexcept;
};

// 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 weak_equality() const noexcept;

Returns: value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent.

constexpr operator partial_ordering() const noexcept;

Returns:

value == 0 ? partial_ordering::equivalent :
value < 0 ? partial_ordering::less :
partial_ordering::greater

constexpr bool operator==(weak_ordering v, unspecified) noexcept;
constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
constexpr bool operator< (weak_ordering v, unspecified) noexcept;
constexpr bool operator> (weak_ordering v, unspecified) noexcept;
constexpr bool operator<= (weak_ordering v, unspecified) noexcept;
constexpr bool operator>= (weak_ordering v, unspecified) noexcept;

Returns: v.value @ 0 for operator@.

constexpr bool operator==(unspecified, weak_ordering v) noexcept;
constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
constexpr bool operator< (unspecified, weak_ordering v) noexcept;
constexpr bool operator> (unspecified, weak_ordering v) noexcept;
constexpr bool operator<= (unspecified, weak_ordering v) noexcept;
constexpr bool operator>= (unspecified, weak_ordering v) noexcept;

Returns: 0 @ v.value for operator@.

constexpr weak_ordering operator<=>(weak_ordering v, unspecified) noexcept;

Returns: v.

constexpr weak_ordering operator<=>(unspecified, weak_ordering v) noexcept;

Returns: v < 0 ? weak_ordering::greater : v > 0 ? weak_ordering::less : v.

17.11.2.6 Class strong_ordering

The strong_ordering type is typically used as the result type of a three-way comparison operator (7.6.8) that (a) admits all of the six two-way comparison operators (7.6.9, 7.6.10), and (b) does imply substitutability.

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;

    } // class strong_ordering

} // namespace std
static const strong_ordering greater;

// conversions
constexpr operator weak_equality() const noexcept;
constexpr operator strong_equality() const noexcept;
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, 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>=(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 bool operator<(unspecified, strong_ordering v) noexcept;
friend constexpr bool operator>(unspecified, strong_ordering v) noexcept;
friend constexpr bool operator<=(unspecified, strong_ordering v) noexcept;
friend constexpr bool operator>=(unspecified, strong_ordering v) noexcept;
friend constexpr strong_ordering operator<=>(strong_ordering v, unspecified) noexcept;
friend constexpr strong_ordering operator<=>(unspecified, strong_ordering v) noexcept;

// valid values' definitions
inline constexpr strong_ordering strong_ordering::less(ord::less);
inline constexpr strong_ordering strong_ordering::equal(eq::equal);
inline constexpr strong_ordering strong_ordering::equivalent(eq::equivalent);
inline constexpr strong_ordering strong_ordering::greater(ord::greater);

constexpr operator weak_equality() const noexcept;
Returns: value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent.

constexpr operator strong_equality() const noexcept;
Returns: value == 0 ? strong_equality::equal : strong_equality::nonequal.

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;
constexpr bool operator>=(strong_ordering v, unspecified) 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;
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: v.value @ 0 for operator@.
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;

Returns: v.
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]

template<class... Ts>
struct common_comparison_category {
  using type = see below;
};

Remarks: The member typedef-name type denotes the common comparison type (11.10.3) of Ts..., the expanded parameter pack. [Note: This is well-defined even if the expansion is empty or includes a type that is not a comparison category type. — end note]

17.11.4 Comparison algorithms

template<class T> constexpr strong_ordering strong_order(const T& a, const T& b);

Effects: Compares two values and produces a result of type strong_ordering:
(1.1) — If numeric_limits<T>::is_iec559 is true, returns a result of type strong_ordering that is consistent with the totalOrder operation as specified in ISO/IEC/IEEE 60559.
(1.2) — Otherwise, returns a <=> b if that expression is well-formed and convertible to strong_ordering.
(1.3) — Otherwise, if the expression a <=> b is well-formed, then the function is defined as deleted.
(1.4) — Otherwise, if the expressions a == b and a < b are each well-formed and convertible to bool, then
  (1.4.1) — if a == b is true, returns strong_ordering::equal;
  (1.4.2) — otherwise, if a < b is true, returns strong_ordering::less;
  (1.4.3) — otherwise, returns strong_ordering::greater.
(1.5) — Otherwise, the function is defined as deleted.

template<class T> constexpr weak_ordering weak_order(const T& a, const T& b);

Effects: Compares two values and produces a result of type weak_ordering:
(2.1) — Returns a <=> b if that expression is well-formed and convertible to weak_ordering.
(2.2) — Otherwise, if the expression a <=> b is well-formed, then the function is defined as deleted.
(2.3) — Otherwise, if the expressions a == b and a < b are each well-formed and convertible to bool, then
  (2.3.1) — if a == b is true, returns weak_ordering::equivalent;
  (2.3.2) — otherwise, if a < b is true, returns weak_ordering::less;
  (2.3.3) — otherwise, returns weak_ordering::greater.
(2.4) — Otherwise, the function is defined as deleted.

template<class T> constexpr partial_ordering partial_order(const T& a, const T& b);

Effects: Compares two values and produces a result of type partial_ordering:
(3.1) — Returns a <=> b if that expression is well-formed and convertible to partial_ordering.
— Otherwise, if the expression \( a \iff b \) is well-formed, then the function is defined as deleted.

— Otherwise, if the expressions \( a == b \) and \( a < b \) are each well-formed and convertible to \( \text{bool} \), then
  
  — if \( a == b \) is \text{true}, returns \text{partial_ordering}::\text{equivalent};
  
  — otherwise, if \( a < b \) is \text{true}, returns \text{partial_ordering}::\text{less};

— otherwise, returns \text{partial_ordering}::\text{greater}.

— Otherwise, the function is defined as deleted.

\[
\text{template<class T> constexpr strong_equality strong_equal(const T& a, const T& b);}\]

\textit{Effects:} Compares two values and produces a result of type \( \text{strong_equality} \):

— Returns \( a \iff b \) if that expression is well-formed and convertible to \( \text{strong_equality} \).

— Otherwise, if the expression \( a \iff b \) is well-formed, then the function is defined as deleted.

— Otherwise, if the expression \( a == b \) is well-formed and convertible to \( \text{bool} \), then
  
  — if \( a == b \) is \text{true}, returns \text{strong_equality}::\text{equal};
  
  — otherwise, returns \text{strong_equality}::\text{nonequal}.

— Otherwise, the function is defined as deleted.

\[
\text{template<class T> constexpr weak_equality weak_equal(const T& a, const T& b);}\]

\textit{Effects:} Compares two values and produces a result of type \( \text{weak_equality} \):

— Returns \( a \iff b \) if that expression is well-formed and convertible to \( \text{weak_equality} \).

— Otherwise, if the expression \( a \iff b \) is well-formed, then the function is defined as deleted.

— Otherwise, if the expression \( a == b \) is well-formed and convertible to \( \text{bool} \), then
  
  — if \( a == b \) is \text{true}, returns \text{weak_equality}::\text{equivalent};
  
  — otherwise, returns \text{weak_equality}::\text{nonequivalent}.

— Otherwise, the function is defined as deleted.

17.12 Coroutines

The header \texttt{<coroutine>} defines several types providing compile and run-time support for coroutines in a C++ program.

17.12.1 Header \texttt{<coroutine>} synopsis

\begin{verbatim}
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 bool operator!=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator<(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator>(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool operator<=(coroutine_handle<> x, coroutine_handle<> y) noexcept;
  constexpr bool 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;
\end{verbatim}
template<> struct coroutine_handle<noop_coroutine_promise>;

using noop_coroutine_handle = coroutine_handle<noop_coroutine_promise>;

noop_coroutine_handle noop_coroutine() noexcept;

// 17.12.5, trivial awaitables
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 satisfies 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 (13.9.2), 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;
        constexpr static 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
        constexpr static coroutine_handle from_address(void* addr);

    };

§ 17.12.3 512
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.

If a program declares an explicit or partial specialization of `coroutine_handle`, the behavior is undefined.

### 17.12.3.1 Construct/reset

```cpp
constexpr coroutine_handle() noexcept;
constexpr coroutine_handle(nullptr_t) noexcept;
```

1. Ensures: `address() == nullptr`.

```cpp
static coroutine_handle from_promise(Promise& p);
```

1. Requires: `p` is a reference to a promise object of a coroutine.
2. Returns: A coroutine handle `h` referring to the coroutine.
3. Ensures: `addressof(h.promise()) == addressof(p)`.

```cpp
coroutine_handle& operator=(nullptr_t) noexcept;
```

1. Ensures: `address() == nullptr`.
2. Returns: `*this`.

### 17.12.3.2 Export/import

```cpp
constexpr void* address() const noexcept;
```

1. Returns: `ptr`.

```cpp
constexpr static coroutine_handle<> coroutine_handle<>::from_address(void* addr);
constexpr static coroutine_handle<Promise> coroutine_handle<>::from_address(void* addr);
```

1. Requires: `addr` was obtained via a prior call to `address`.
2. Ensures: `from_address(address()) == *this`.

### 17.12.3.3 Observers

```cpp
constexpr explicit operator bool() const noexcept;
```

1. Returns: `address() != nullptr`.

```cpp
bool done() const;
```

1. Requires: `*this` refers to a suspended coroutine.
2. 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 is either an instance of `std::thread` or 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]

```cpp
void operator()() const;
void resume() const;
```

1. Requires: `*this` refers to a suspended coroutine.
2. Effects: Resumes the execution of the coroutine. If the coroutine was suspended at its final suspend point, behavior is undefined.
void destroy() const;

4 Requires: *this refers to a suspended coroutine.
5 Effects: Destroys the coroutine (9.4.4).

17.12.3.5 Promise access [coroutine.handle.promise]

Promise& promise() const;
1 Requires: *this refers to a coroutine.
2 Returns: A reference to the promise of the coroutine.

17.12.3.6 Comparison operators [coroutine.handle.compare]

constexpr bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
1 Returns: x.address() == y.address().

constexpr bool operator<(coroutine_handle<> x, coroutine_handle<> y) noexcept;
2 Returns: less<>()(x.address(), y.address()).

17.12.3.7 Hash support [coroutine.handle.hash]

template<class P> struct hash<coroutine_handle<P>>;
1 The specialization is enabled (20.14.18).

17.12.4 No-op coroutines [coroutine.noop]

17.12.4.1 Class noop_coroutine_promise [coroutine.promise.noop]

struct noop_coroutine_promise {};
1 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> [coroutine.handle.noop]

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;

            // 17.12.4.2.2, resumption
            constexpr void operator()() const noexcept;
            constexpr void resume() const noexcept;
            constexpr void destroy() const noexcept;

            // 17.12.4.2.3, promise access
            noop_coroutine_promise& promise() const noexcept;

            // 17.12.4.2.4, 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;
1 Returns: true.

constexpr bool done() const noexcept;
2 Returns: false.

§ 17.12.4.2.1 514
17.12.4.2.2 Resumption

constexpr void operator()() const noexcept;  
constexpr void resume() const noexcept;  
constexpr void destroy() const noexcept;

1 Effects: None.
2 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

noop_coroutine_promise& promise() const noexcept;

1 Returns: A reference to the promise object associated with this coroutine handle.

17.12.4.2.4 Address

constexpr void* address() const noexcept;

1 Returns: ptr.
2 Remarks: A noop_coroutine_handle’s ptr is always a non-null pointer value.

17.12.4.3 Function noop_coroutine

noop_coroutine_handle noop_coroutine() noexcept;

1 Returns: A handle to a coroutine that has no observable effects when resumed or destroyed.
2 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

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 {}  
};
}

1 [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

1 Headers <setjmp> (nonlocal jumps), <signal> (signal handling), <stdarg> (variable arguments), and <stdlib> (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 (16.5.5.9) 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

namespace std {
  using va_list = see below;
}
#define va_arg(V, P) see below
#define va_copy(VDST, VSRC) see below
#define va_end(V) see below
#define va_start(V, P) see below

The contents of the header <stdarg> are the same as the C standard library header <stdarg.h>, with the following changes: The restrictions that ISO C places on the second parameter to the va_start macro in header <stdarg.h> are different in this document. The parameter parmN is the rightmost parameter in the variable parameter list of the function definition (the one just before the ...). If the parameter parmN is a pack expansion (13.6.3) 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

namespace std {
  using jmp_buf = see below;
  [[noreturn]] void longjmp(jmp_buf env, int val);
}
#define setjmp(env) see below

The contents of the header <csetjmp> are the same as the C standard library header <setjmp.h>.

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 automatic objects. 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

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);

  int raise(int sig);
}

#define SIG_DFL see below
#define SIG_ERR see below
#define SIG_IGN see below
#define SIGABRT see below
#define SIGFPE see below
#define SIGILL see below
#define SIGINT see below
#define SIGSEGV see below
#define SIGTERM see below

The contents of the header <csignal> are the same as the C standard library header <signal.h>.

17.13.4 Signal handlers

A call to the function signal synchronizes with any resulting invocation of the signal handler so installed.

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

Note that va_start is required to work as specified even if unary operator& is overloaded for the type of parmN.
f is a non-member function, and for every pointer-to-atomic argument A passed to f, atomic_is_lock_free(A) yields true.

An evaluation is signal-safe unless it includes one of the following:

- 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]

- an access to an object with thread storage duration;

- a dynamic_cast expression;

- throwing of an exception;

- control entering a try-block or function-try-block;

- initialization of a variable with static storage duration requiring dynamic initialization (6.8.3.3, 8.7); or

- waiting for the completion of the initialization of a variable with static storage duration (8.7).

A signal handler invocation has undefined behavior if it includes an evaluation that is not signal-safe.

The function signal is signal-safe if it is invoked with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler.

See also: ISO C 7.14

---

221) Such initialization might occur because it is the first odr-use (6.2) 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 37.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2 Equality preservation</td>
<td></td>
</tr>
<tr>
<td>18.4 Language-related concepts</td>
<td>&lt;concepts&gt;</td>
</tr>
<tr>
<td>18.5 Comparison concepts</td>
<td></td>
</tr>
<tr>
<td>18.6 Object concepts</td>
<td></td>
</tr>
<tr>
<td>18.7 Callable concepts</td>
<td></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.

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.

Where a *requires-expression* declares an expression that is non-modifying for some constant lvalue operand, additional variations of that expression that accept a non-constant lvalue or (possibly constant) rvalue for the given operand are also required except where such an expression variation is explicitly required with differing semantics. These *implicit expression variations* are required to meet the semantic requirements of the declared expression. The extent to which an implementation validates the syntax of the variations is unspecified.

[Example:

```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

\[
\begin{align*}
& c == \text{std::move}(d); & & c == \text{std::move}(b); \\
& \text{std::move}(c) == d; & & \text{std::move}(c) == b; \\
& \text{std::move}(c) == \text{std::move}(d); & & \text{std::move}(c) == \text{std::move}(b); \\
& a == d; & & a == b; \\
& a == \text{std::move}(d); & & a == \text{std::move}(b); \\
& \text{std::move}(a) == d; & & \text{std::move}(a) == b; \\
& \text{std::move}(a) == \text{std::move}(d); & & \text{std::move}(a) == \text{std::move}(b);
\end{align*}
\]

had been 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 = \text{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]

[Example: The following type T meets the explicitly stated syntactic requirements of concept C above but does not meet the additional implicit requirements:

\[
\text{struct T} \\
\begin{align*}
& \text{bool operator==}(\text{const T}&) \text{ const} \{ \text{return true;} \} \\
& \text{bool operator==}(T&) = \text{delete}; \\
\end{align*}
\]

T fails to meet the implicit requirements of C, so C<T> is not satisfied. 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

namespace std {
  // 18.4, language-related concepts
  // 18.4.2, concept Same
  template<class T, class U>
  concept Same = see below;

  // 18.4.3, concept DerivedFrom
  template<class Derived, class Base>
  concept DerivedFrom = see below;

  // 18.4.4, concept ConvertibleTo
  template<class From, class To>
  concept ConvertibleTo = see below;

  // 18.4.5, concept CommonReference
  template<class T, class U>
  concept CommonReference = see below;

  // 18.4.6, concept Common
  template<class T, class U>
  concept Common = see below;

  // 18.4.7, integral concepts
  template<class T>
  concept Integral = see below;

§ 18.3
template<class T>
    concept SignedIntegral = see below;
template<class T>
    concept UnsignedIntegral = see below;

// 18.4.8, concept Assignable
template<class LHS, class RHS>
    concept Assignable = 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 SwappableWith = see below;

// 18.4.10, concept Destructible
template<class T>
    concept Destructible = see below;

// 18.4.11, concept Constructible
template<class T, class... Args>
    concept Constructible = see below;

// 18.4.12, concept DefaultConstructible
template<class T>
    concept DefaultConstructible = see below;

// 18.4.13, concept MoveConstructible
template<class T>
    concept MoveConstructible = see below;

// 18.4.14, concept CopyConstructible
template<class T>
    concept CopyConstructible = see below;

// 18.5, comparison concepts
// 18.5.2, concept Boolean
template<class B>
    concept Boolean = see below;

// 18.5.3, concept EqualityComparable
template<class T>
    concept EqualityComparable = see below;
template<class T, class U>
    concept EqualityComparableWith = see below;

// 18.5.4, concept StrictTotallyOrdered
template<class T>
    concept StrictTotallyOrdered = see below;
template<class T, class U>
    concept StrictTotallyOrderedWith = see below;

// 18.6, object concepts
template<class T>
    concept Movable = see below;
template<class T>
    concept Copyable = see below;
template<class T>
    concept Semiregular = see below;
template<class T>
concept Regular = see below;

// 18.7, callable concepts
// 18.7.2, concept Invocable
template<class F, class... Args>
concept Invocable = see below;

// 18.7.3, concept RegularInvocable
template<class F, class... Args>
concept RegularInvocable = see below;

// 18.7.4, concept Predicate
template<class F, class... Args>
concept Predicate = see below;

// 18.7.5, concept Relation
template<class R, class T, class U>
concept Relation = see below;

// 18.7.6, concept StrictWeakOrder
template<class R, class T, class U>
concept StrictWeakOrder = see below;
}

18.4 Language-related concepts

18.4.1 General

This subclause 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

template<class T, class U>
concept same-impl = is_same_v<T, U>; // exposition only

template<class T, class U>
concept Same = same-impl<T, U> && same-impl<U, T>;

[Note: Same<T, U> subsumes Same<U, T> and vice versa. — end note]

18.4.3 Concept DerivedFrom

template<class Derived, class Base>
concept DerivedFrom =
is_base_of_v<Base, Derived> &&
is_convertible_v<const volatile Derived*, const volatile Base*>;

[Note: DerivedFrom<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 ConvertibleTo

The ConvertibleTo concept requires an expression of a particular type and value category to be both implicitly and explicitly convertible to some other type. The implicit and explicit conversions are required to produce equal results.

template<class From, class To>
concept ConvertibleTo =
is_convertible_v<From, To> &&
requires(From (&f)()) {
    static_cast<To>(f());
};

Let test be the invented function:
To test(From (&f)()) {
    return f();
}

and let f be a function with no arguments and return type From such that f() is equality-preserving. ConvertibleTo<From, To> is satisfied only if:

1. To is not an object or reference-to-object type, or static_cast<To>(f()) is equal to test(f).
2. From is not a reference-to-object type, or
   - If From is a raw value reference to a non const-qualified type, converting f() to To is valid and unspecified (16.5.5.15).
   - Otherwise, f() is not modified by either expression.

18.4.5 Concept CommonReference

For two types T and U, if common_reference_t<T, U> is well-formed and denotes a type C such that both ConvertibleTo<T, C> and ConvertibleTo<U, C> are satisfied, 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]

template<class T, class U>
concept CommonReference =
    Same<common_reference_t<T, U>, common_reference_t<U, T>> &&
    ConvertibleToT, common_reference_t<T, U>> &&
    ConvertibleToU, 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 CommonReference<T, U> only if:

1. C(t1) equals C(t2) if and only if t1 equals t2, and
2. C(u1) equals C(u2) if and only if u1 equals u2.

[Note: Users can customize the behavior of CommonReference by specializing the basic_common_reference class template (20.15.7.6). — end note]

18.4.6 Concept Common

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]

template<class T, class U>
concept Common =
    Same<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>());
    } &&
    CommonReference<
        add_lvalue_reference_t<const T>,
        add_lvalue_reference_t<const U>> &&
    CommonReference<
        add_lvalue_reference_t<const common_type_t<T, U>>,
        common_reference_t<
            add_lvalue_reference_t<const T>,
            add_lvalue_reference_t<const U>>;
18.4.7 Integral concepts

```
template<class T>
concept Integral = is_integral_v<T>;

template<class T>
concept SignedIntegral = Integral<T> && is_signed_v<T>;

template<class T>
concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

1. \[Note: \] SignedIntegral<T> can be satisfied even by types that are not signed integral types (6.7.1); for example, char. —end note

2. \[Note: \] UnsignedIntegral<T> can be satisfied even by types that are not unsigned integral types (6.7.1); for example, bool. —end note

18.4.8 Concept Assignable

```
template<class LHS, class RHS>
concept Assignable =
  is_lvalue_reference_v<LHS> &&
  CommonReference<const remove_reference_t<LHS>&, const remove_reference_t<RHS>&> &&
  requires(LHS lhs, RHS&& rhs) {
    ( lhs = std::forward<RHS>(rhs) ) -> Same<LHS>;
  }
```

1. Let:
   1.1. \( lhs \) be an lvalue that refers to an object \( lcopy \) such that \( \text{decltype}((\text{lhs})) \) is \( \text{LHS} \),
   1.2. \( rhs \) be an expression such that \( \text{decltype}((\text{rhs})) \) is \( \text{RHS} \), and
   1.3. \( rcopy \) be a distinct object that is equal to \( \text{rhs} \).

2. Assignable<LHS, RHS> is satisfied only if
   1.4. addressof(lhs = rhs) == addressof(lcopy).
   1.5. After evaluating \( lhs = rhs \):
      1.5.1. \( lhs \) is equal to \( rcopy \), unless \( rhs \) is a non-const xvalue that refers to \( lcopy \).
      1.5.2. If \( rhs \) is a non-const xvalue, the resulting state of the object to which it refers is valid but unspecified (16.5.5.15).
      1.5.3. Otherwise, if \( rhs \) is a glvalue, the object to which it refers is not modified.

18.4.9 Concept Swappable

1. Let \( t1 \) and \( t2 \) be equality-preserving expressions that denote distinct equal objects of type \( T \), and let \( u1 \) and \( u2 \) similarly denote distinct equal objects of type \( U \). \[Note: \] \( t1 \) and \( u1 \) can denote distinct objects, or the same object. —end note An operation \textit{exchanges the values} denoted by \( t1 \) and \( u1 \) if and only if the operation modifies neither \( t2 \) nor \( u2 \) and:
   1.1. If \( T \) and \( U \) are the same type, the result of the operation is that \( t1 \) equals \( u2 \) and \( u1 \) equals \( t2 \).
   1.2. If \( T \) and \( U \) are different types that model \text{CommonReference<const Tk, const Uk>}, the result of the operation is that \( C(t1) \) equals \( C(u2) \) and \( C(u1) \) equals \( C(t2) \) where \( C \) is \text{common_reference_t<const Tk, const Uk>}.

2. The name \texttt{ranges::swap()} denotes a customization point object (16.4.2.2.6). The expression \texttt{ranges::swap(E1, E2)} for some subexpressions \( E1 \) and \( E2 \) is expression-equivalent to an expression \( S \) determined as follows:
   2.1. \( S \) is \((\text{void})\text{swap}(E1, E2)\) if \( E1 \) or \( E2 \) has class or enumeration type (6.7.2) and that expression is valid, with overload resolution performed in a context that includes the declarations

222) The name \texttt{swap} is used here unqualified.
template<class T>
void swap(T& a, T& b) = delete;

template<class T, size_t N>
void swap(T&(N), T&(N)) = delete;

and does not include a declaration of `ranges::swap`. If the function selected by overload resolution does not exchange the values denoted by `E1` and `E2`, the program is ill-formed with no diagnostic required.

(2.2) Otherwise, if `E1` and `E2` are lvalues of array types (6.7.2) with equal extent and `ranges::swap(*E1, *E2)` is a valid expression, `S` is `(void)ranges::swap_ranges(E1, E2)`, except that `noexcept(S)` is equal to `noexcept(ranges::swap(*E1, *E2))`.

(2.3) Otherwise, if `E1` and `E2` are lvalues of the same type `T` that models `MoveConstructible<T>` and `Assignable<T, T>`, `S` is an expression that exchanges the denoted values. `S` is a constant expression if

(2.3.1) `T` is a literal type (6.7),
(2.3.2) both `E1 = std::move(E2)` and `E2 = std::move(E1)` are constant subexpressions (16.3.6), and
(2.3.3) the full-expressions of the initializers in the declarations

\[
\begin{align*}
T \ t1 &= std::move(E1); \\
T \ t2 &= std::move(E2);
\end{align*}
\]

are constant subexpressions.

`noexcept(S)` is equal to `is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>`.

(2.4) Otherwise, `ranges::swap(E1, E2)` is ill-formed. [Note: This case can result in substitution failure when `ranges::swap(E1, E2)` appears in the immediate context of a template instantiation. — end note]

3 [Note: Whenever `ranges::swap(E1, E2)` is a valid expression, it exchanges the values denoted by `E1` and `E2` and has type `void`. — end note]

```
template<class T>
concept Swappable = requires(T& a, T& b) { ranges::swap(a, b); };

template<class T, class U>
concept SwappableWith =
    CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
    requires(T&& t, U&& u) { 
        ranges::swap(std::forward<T>(t), std::forward<U>(u));
        ranges::swap(std::forward<T>(u), std::forward<U>(t));
    };
```

4 [Note: The semantics of the `Swappable` and `SwappableWith` concepts are fully defined by the `ranges::swap` customization point. — end note]

5 [Example: User code can ensure that the evaluation of `swap` calls is performed in an appropriate context under the various conditions as follows:

```
#include <cassert>
#include <concepts>
#include <utility>

namespace ranges = std::ranges;

template<class T, std::SwappableWith<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 proxy(A& a) { return Proxy{ &a }; }

  void swap(A& x, Proxy p) {
    ranges::swap(x.m, p.a->m);
  }

  void swap(Proxy p, &x) { swap(x, p); } // satisfy symmetry requirement
}

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]
Constructible<T, const T> && ConvertibleTo<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. CopyConstructible<T> is satisfied only if

(1.1) After the definition T u = v;, u is equal to v.

(1.2) T(v) is equal to v.

18.5 Comparison concepts

18.5.1 General

This subclause describes concepts that establish relationships and orderings on values of possibly differing object types.

18.5.2 Concept Boolean

The Boolean concept specifies the requirements on a type that is usable in Boolean contexts.

```cpp
template<class B>
concept Boolean =
    Movable<remove_cvref_t<B>> && // (see 18.6)
    requires(const remove_reference_t<B>& b1,
             const remove_reference_t<B>& b2, const bool a) {
        requires ConvertibleTo<const remove_reference_t<B>&, bool>;
        { !b1 } -> ConvertibleTo<bool>;
        { b1 && a } -> Same<bool>;
        { b1 || a } -> Same<bool>;
        { b1 && b2 } -> Same<bool>;
        { a && b2 } -> Same<bool>;
        { b1 || b2 } -> Same<bool>;
        { a || b2 } -> Same<bool>;
        { b1 == b2 } -> ConvertibleTo<bool>;
        { b1 == a } -> ConvertibleTo<bool>;
        { a == b2 } -> ConvertibleTo<bool>;
        { b1 != b2 } -> ConvertibleTo<bool>;
        { a != b2 } -> ConvertibleTo<bool>;
    };
```

Let b1 and b2 be lvalues of type const remove_reference_t<B>. Boolean<B> is satisfied only if

(2.1) bool(b1) == !bool(!b1).

(2.2) (b1 && b2), (b1 && bool(b2)), and (bool(b1) && b2) are all equal to (bool(b1) && bool(b2)), and have the same short-circuit evaluation.

(2.3) (b1 || b2), (b1 || bool(b2)), and (bool(b1) || b2) are all equal to (bool(b1) || bool(b2)), and have the same short-circuit evaluation.

(2.4) bool(b1 == b2), bool(b1 == bool(b2)), and bool(bool(b1) == b2) are all equal to (bool(b1) == bool(b2)).

(2.5) bool(b1 != b2), bool(b1 != bool(b2)), and bool(bool(b1) != b2) are all equal to (bool(b1) != bool(b2)).

[Example: The types bool, true_type (20.15.2), and bitset<N>::reference (20.9.2) are Boolean types. Pointers, smart pointers, and types with only explicit conversions to bool are not Boolean types. — end example]

18.5.3 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;
        { t != u } -> Boolean;
        { u == t } -> Boolean;
    };
```
Let \( t \) and \( u \) be lvalues of types `const remove_reference_t<T>` and `const remove_reference_t<U>` respectively. `weakly-equality-comparable-with<T, U>` is satisfied 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 EqualityComparable = weakly-equality-comparable-with<T, T>;
```

Let \( a \) and \( b \) be objects of type \( T \). `EqualityComparable<T>` is satisfied only if \( \text{bool}(a == b) \) is true when \( a \) is equal to \( b \) (18.2), and false otherwise.

```
template<class T, class U>
    concept EqualityComparableWith =
        EqualityComparable<T> && EqualityComparable<U> &&
        CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
        EqualityComparable<
            common_reference_t<const remove_reference_t<T>&, const remove_reference_t<U>&>;
```

18.5.4 Concept StrictTotallyOrdered

```
template<class T>
    concept StrictTotallyOrdered =
        EqualityComparable<T> &&
        requires(const remove_reference_t<T>& a, const remove_reference_t<T>& b) {
            { a < b } -> Boolean;
            { a > b } -> Boolean;
            { a <= b } -> Boolean;
            { a >= b } -> Boolean;
        };
```

Let \( a, b, \) and \( c \) be lvalues of type `const remove_reference_t<T>`.

\( \text{StrictTotallyOrdered<T>} \) is satisfied only if:

1. \( \text{Exactly one of bool}(a < b), \text{bool}(a > b), \) or \( \text{bool}(a == b) \) is true.
2. \( \text{If bool}(a < b) \) and \( \text{bool}(b < c) \), then \( \text{bool}(a < c) \).
3. \( \text{bool}(a > b) == \text{bool}(b < a) \).
4. \( \text{bool}(a <= b) == !\text{bool}(b < a) \).
5. \( \text{bool}(a >= b) == !\text{bool}(a < b) \).

```
template<class T, class U>
    concept StrictTotallyOrderedWith =
        StrictTotallyOrdered<T> && StrictTotallyOrdered<U> &&
        CommonReference<const remove_reference_t<T>&, const remove_reference_t<U>&> &&
        StrictTotallyOrdered<
            common_reference_t<const remove_reference_t<T>&, const remove_reference_t<T>&>;
```

§ 18.5.4
const remove_reference_t<T>& &&
EqualityComparableWith<T, U>&&
requires(const remove_reference_t<T>& t,
const remove_reference_t<U>& u) {
    { t < u }  -> Boolean;
    { t > u }  -> Boolean;
    { t <= u } -> Boolean;
    { t >= u } -> Boolean;
    { u < t }  -> Boolean;
    { u > t }  -> Boolean;
    { u <= t } -> Boolean;
    { u >= t } -> Boolean;
};

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>&>

StrictTotallyOrderedWith<T, U> is satisfied only if

- \( \text{bool}(t < u) = \text{bool}(C(t) < C(u)) \),
- \( \text{bool}(t > u) = \text{bool}(C(t) > C(u)) \),
- \( \text{bool}(t <= u) = \text{bool}(C(t) <= C(u)) \),
- \( \text{bool}(t >= u) = \text{bool}(C(t) >= C(u)) \),
- \( \text{bool}(u < t) = \text{bool}(C(u) < C(t)) \),
- \( \text{bool}(u > t) = \text{bool}(C(u) > C(t)) \),
- \( \text{bool}(u <= t) = \text{bool}(C(u) <= C(t)) \),
- \( \text{bool}(u >= t) = \text{bool}(C(u) >= C(t)) \).

18.6 Object concepts

This subclause describes concepts that specify the basis of the value-oriented programming style on which the library is based.

```cpp
template<class T>
concept Movable = is_object_v<T> && MoveConstructible<T> && Assignable<T&, T> && Swappable<T>;

template<class T>
concept Copyable = CopyConstructible<T> && Movable<T> && Assignable<T&, const T&>;

template<class T>
concept Semiregular = Copyable<T> && DefaultConstructible<T>;

template<class T>
concept Regular = Semiregular<T> && EqualityComparable<T>;
```

[Note: The `Semiregular` concept is satisfied by types that behave similarly to built-in types like `int`, except that they might not be comparable with `==`. — end note]

[Note: The `Regular` concept is satisfied 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

The concepts in this subclause describe the requirements on function objects (20.14) and their arguments.

18.7.2 Concept Invocable

The `Invocable` concept specifies a relationship between a callable type (20.14.2) \( F \) and a set of argument types \( \text{Args}... \) which can be evaluated by the library function \( \text{invoke} \) (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 satisfy Invocable, since the invoke function call expression is not required to be equality-preserving (18.2). —end example]

18.7.3 Concept RegularInvocable

```cpp
template<class F, class... Args>
concept RegularInvocable = Invocable<F, Args...>;
```

1 The invoke function call expression shall be equality-preserving and shall not modify the function object or the arguments (18.2). [Note: This requirement supersedes the annotation in the definition of Invocable. —end note]

2 Example: A random number generator does not satisfy RegularInvocable. —end example]

3 [Note: The distinction between Invocable and RegularInvocable is purely semantic. —end note]

18.7.4 Concept Predicate

```cpp
template<class F, class... Args>
concept Predicate = RegularInvocable<F, Args...> && Boolean<invoke_result_t<F, Args...>>;
```

18.7.5 Concept Relation

```cpp
template<class R, class T, class U>
concept Relation =
Predicate<R, T, T> && Predicate<R, U, U> &&
Predicate<R, T, U> && Predicate<R, U, T>;
```

18.7.6 Concept StrictWeakOrder

```cpp
template<class R, class T, class U>
concept StrictWeakOrder = Relation<R, T, U>;
```

1 A Relation satisfies StrictWeakOrder only if it imposes a strict weak ordering on its arguments.

2 The term strict refers to the requirement of an irreflexive relation (\(!\text{comp}(x, x)\) 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 \(!\text{comp}(a, b) && !\text{comp}(b, a)\), then the requirements are that comp and equiv both be transitive relations:

\[(2.1)\]
\[\text{comp}(a, b) && \text{comp}(b, c) \implies \text{comp}(a, c)\]

\[(2.2)\]
\[\text{equiv}(a, b) && \text{equiv}(b, c) \implies \text{equiv}(a, c)\]

3 [Note: Under these conditions, it can be shown that

\[(3.1)\]
\[\text{equiv} \text{ is an equivalence relation.}\]

\[(3.2)\]
\[\text{comp} \text{ induces a well-defined relation on the equivalence classes determined by equiv, and}\]

\[(3.3)\]
\[\text{the induced relation is a strict total ordering.}\]

—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 38.

Table 38 — Diagnostics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.2 Exception classes</td>
<td>&lt;stdexcept&gt;</td>
</tr>
<tr>
<td>19.3 Assertions</td>
<td>&lt;cassert&gt;</td>
</tr>
<tr>
<td>19.4 Error numbers</td>
<td>&lt;cerrno&gt;</td>
</tr>
<tr>
<td>19.5 System error support</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 (16.5.5.12) 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. 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);

Ensures: strcmp(what(), what_arg.c_str()) == 0.
logic_error(const char* what_arg);

Ensures: 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);
    };
}

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);

Ensures: strcmp(what(), what_arg.c_str()) == 0.

domain_error(const char* what_arg);

Ensures: 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);
    };
}

The class invalid_argument defines the type of objects thrown as exceptions to report an invalid argument.

invalid_argument(const string& what_arg);

Ensures: strcmp(what(), what_arg.c_str()) == 0.

invalid_argument(const char* what_arg);

Ensures: 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);
    };
}

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);

Ensures: strcmp(what(), what_arg.c_str()) == 0.

length_error(const char* what_arg);

Ensures: 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);
```

Ensures: `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
out_of_range(const char* what_arg);
```

Ensures: `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);
```

Ensures: `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
runtime_error(const char* what_arg);
```

Ensures: `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);
```

Ensures: `strcmp(what(), what_arg.c_str()) == 0`.

```cpp
range_error(const char* what_arg);
```

Ensures: `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);
```

Ensures: `strcmp(what(), what_arg.c_str()) == 0`.

§ 19.2.9
overflow_error(const char* what_arg);

Ensures: 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);
    };
}

The class underflow_error defines the type of objects thrown as exceptions to report an arithmetic underflow error.

underflow_error(const string& what_arg);

Ensures: strcmp(what(), what_arg.c_str()) == 0.

underflow_error(const char* what_arg);

Ensures: 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.

19.3.1 Header <cassert> synopsis

#define assert(E) see below

The contents are the same as the C standard library header <assert.h>, except that a macro named static_assert is not defined.

SEE ALSO: ISO C 7.2

19.3.2 The assert macro

An expression assert(E) is a constant subexpression (16.3.6), if

— NDEBUG is defined at the point where assert is last defined or redefined, or
— 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 E2BIG see below
#define EACCES see below
#define EADDRINUSE see below
#define EADDRNOTAVAIL see below
#define EAFNOSUPPORT see below
#define EAGAIN see below
#define EALREADY see below
#define EBADF see below
#define EBADMSG see below
#define EBUSY see below
#define ECANCELED see below
#define ECHILD see below
#define ECONNABORTED see below
#define ECONNREFUSED see below
#define ECONNRESET see below
#define ECANCELED see below
#define ECHILD see below
#define ECONNABORTED see below
#define ECONNREFUSED see below
#define ECONNRESET see below
#define EDEADLK see below
#define EDESTADDRREQ see below
#define EDOM see below
#define EXIST see below
#define EFAULT see below
#define EFBIG see below
#define EHOSTUNREACH 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 ENAME_TOO_LONG see below
#define ENETDOWN see below
#define EINETRESET 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 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 ENOPTY 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 ERFS see below
#define EPIPE see below
#define EPROTOTYPE see below
#define EPROTONOSUPPORT see below
#define EPROTOTYPE see below
#define ERANGE see below
#define EROFS see below
#define ESPIPE see below
#define ESRCH see below
#define ETIME see below
#define ETIMEDOUT see below
#define ETXTBSY see below
#define EWOULDBLOCK see below
#define EXDEV see below

§ 19.4.1 534
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

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.

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

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 {};

denum class errc {
    address_family_not_supported, // EAFNSUPPORT
    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, // EXIST
    file_too_large, // E2BIG
    filename_too_long, // ENAMETOOLONG
    function_not_supported, // ENOSYS
    host_unreachable, // EHOSTUNREACH
    identifier_removed, // EIDRM
    illegal_byte_sequence, // EILSEQ
    inappropriate_io_control_operation, // ENOTSYS
    interrupted, // EINTR
    invalid_argument, // EINVAL
    invalid_seek, // ESPICE
    io_error, // EIO
    is_a_directory, // EISDIR
    message_size, // EMSGSIZE
    network_down, // ENETDOWN
    network_reset, // ENETRESET
    network_unreachable, // ENETUNREACH

§ 19.5.1
no_buffer_space, // ENOBUFS
no_child_process, // ECHILD
no_link, // ENOLINK
no_lock_available, // ENOLCK
no_message_available, // ENODATA
no_message, // ENOMSG
no_protocol_option, // ENOPROTOOPT
no_space_on_device, // ENOSPC
no_stream_resources, // ENOSR
no_such_device_or_address, // ENXIO
no_such_device, // ENODEV
no_such_file_or_directory, // ENOENT
no_such_process, // ESRCH
not_a_directory, // ENOTDIR
not_a_stream, // ENOSES
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, // EACCESS
protocol_error, // EPROTONOSUPPORT
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_code& rhs) noexcept;
bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
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_code& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;

§ 19.5.1
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.

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 [syserr.errcat]

19.5.2.1 Overview [syserr.errcat.overview]

The class error_category serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from error_category to support categories of errors in addition to those defined in this document. Such classes shall behave as specified in this 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;
    bool operator!=(const error_category& rhs) const noexcept;
    bool 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 [syserr.errcat.virtuals]

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;

Returns: default_error_condition(code) == condition.

template<typename Categories>
virtual bool equivalent(Categories categories) const noexcept;

Returns: *this == categories

virtual bool equivalent(const error_code& code, int condition) const noexcept;

Returns: *(this == code.category() && code.value()) == condition.

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.

bool operator!=(const error_category& rhs) const noexcept;

Returns: !(*this == rhs).

bool operator<(const error_category& rhs) const noexcept;

Returns: less<const error_category*>(this, &rhs).

[Note: less (20.14.7) 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]

```cpp
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

```cpp
error_code() noexcept;
```

1 Ensures: val_ == 0 and cat_ == &system_category().

```cpp
error_code(int val, const error_category& cat) noexcept;
```

2 Ensures: val_ == val and cat_ == &cat.

```cpp
template<class ErrorCodeEnum>
error_code(ErrorCodeEnum e) noexcept;
```

3 Constraints: is_error_code_enum_v<ErrorCodeEnum> is true.

4 Ensures: *this == make_error_code(e).

19.5.3.3 Modifiers

```cpp
void assign(int val, const error_category& cat) noexcept;
```

1 Ensures: val_ == val and cat_ == &cat.

```cpp
template<class ErrorCodeEnum>
error_code& operator=(ErrorCodeEnum e) noexcept;
```

2 Constraints: is_error_code_enum_v<ErrorCodeEnum> is true.
Ensures: \*this == make_error_code(e).

Returns: \*this.

void clear() noexcept;

Ensures: value() == 0 and category() == system_category().

19.5.3.4 Observers

int value() const noexcept;

Returns: val_.

const error_category& category() const noexcept;

Returns: *cat_.

error_condition default_error_condition() const noexcept;

Returns: category().default_error_condition(value()).

string message() const;

Returns: category().message(value()).

explicit operator bool() const noexcept;

Returns: value() != 0.

19.5.3.5 Non-member functions

error_code make_error_code(errc e) noexcept;

Returns: error_code(static_cast<int>(e), generic_category()).

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& os, const error_code& ec);

Effects: Equivalent to: return os << ec.category().name() << ':' << ec.value();

19.5.4 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

    error_condition() noexcept;
Ensures: val_ == 0 and cat_ == &generic_category().
    error_condition(int val, const error_category& cat) noexcept;
Ensures: val_ == val and cat_ == &cat.

    template<class ErrorConditionEnum>
    error_condition(ErrorConditionEnum e) noexcept;
Constraints: is_error_condition_enum_v<ErrorConditionEnum> is true.
Ensures: *this == make_error_condition(e).

19.5.4.3 Modifiers

    void assign(int val, const error_category& cat) noexcept;
Ensures: val_ == val and cat_ == &cat.

    template<class ErrorConditionEnum>
    error_condition& operator=(ErrorConditionEnum e) noexcept;
Constraints: is_error_condition_enum_v<ErrorConditionEnum> is true.
Ensures: *this == make_error_condition(e).
Returns: *this.

    void clear() noexcept;
Ensures: value() == 0 and category() == generic_category().

19.5.4.4 Observers

    int value() const noexcept;
Returns: val_.

    const error_category& category() const noexcept;
Returns: *cat_.

    string message() const;
Returns: category().message(value()).

    explicit operator bool() const noexcept;
Returns: value() != 0.

19.5.4.5 Non-member functions

    error_condition make_error_condition(errc e) noexcept;
Returns: error_condition(static_cast<int>(e), generic_category()).

19.5.5 Comparison functions

    bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
Returns:
    lhs.category() == rhs.category() && lhs.value() == rhs.value()
bool operator==(const error_code& lhs, const error_condition& rhs) noexcept;

Returns:
    lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value())

bool operator==(const error_condition& lhs, const error_code& rhs) noexcept;

Returns:
    rhs.category().equivalent(rhs.value(), lhs) || lhs.category().equivalent(rhs, lhs.value())

bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;

Returns:
    lhs.category() == rhs.category() && lhs.value() == rhs.value()  

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_code& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;

Returns: ! (lhs == rhs).

bool operator<(const error_code& lhs, const error_code& rhs) noexcept;

Returns:
    lhs.category() < rhs.category() ||
    (lhs.category() == rhs.category() && lhs.value() < rhs.value())

bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;

Returns:
    lhs.category() < rhs.category() ||
    (lhs.category() == rhs.category() && 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);
            const error_code& code() const noexcept;
            const char* what() const noexcept override;
        }
    }
}
19.5.7.2 Members

system_error(error_code ec, const string& what_arg);

1 Ensures: code() == ec and
string_view(what()).find(what_arg.c_str()) != string_view::npos.

system_error(error_code ec, const char* what_arg);

2 Ensures: code() == ec and string_view(what()).find(what_arg) != string_view::npos.

system_error(error_code ec);

3 Ensures: code() == ec.

system_error(int ev, const error_category& ecat, const string& what_arg);

4 Ensures: 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 Ensures: code() == error_code(ev, ecat) and
string_view(what()).find(what_arg) != string_view::npos.

system_error(int ev, const error_category& ecat);

6 Ensures: 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 [utilities]

20.1 General [utilities.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 39.

Table 39 — General utilities library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.2</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.3</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.4</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>20.5</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>20.6</td>
<td>&lt;optional&gt;</td>
</tr>
<tr>
<td>20.7</td>
<td>&lt;variant&gt;</td>
</tr>
<tr>
<td>20.8</td>
<td>&lt;any&gt;</td>
</tr>
<tr>
<td>20.9</td>
<td>&lt;bitset&gt;</td>
</tr>
<tr>
<td>20.10</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>20.11</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>20.12</td>
<td>&lt;memory_resource&gt;</td>
</tr>
<tr>
<td>20.13</td>
<td>&lt;scoped_allocator&gt;</td>
</tr>
<tr>
<td>20.14</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>20.15</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>20.16</td>
<td>&lt;ratio&gt;</td>
</tr>
<tr>
<td>20.17</td>
<td>&lt;typeindex&gt;</td>
</tr>
<tr>
<td>20.18</td>
<td>&lt;execution&gt;</td>
</tr>
<tr>
<td>20.19</td>
<td>&lt;charconv&gt;</td>
</tr>
</tbody>
</table>

20.2 Utility components [utility]

20.2.1 Header <utility> synopsis [utility.syn]

The header <utility> contains some basic function and class templates that are used throughout the rest of the library.

```
#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 remove_reference_t<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 add_const_t<T>& as_const(T& t) noexcept;
template<class T>
void as_const(const T&&) = delete;

// 20.2.6, declval
template<class T>
add_rvalue_reference_t<T> declval() noexcept; // as unevaluated operand

// 20.3, Compile-time integer sequences
template<class T, T...>
struct integer_sequence;
template<size_t... I>
using index_sequence = integer_sequence<size_t, I...>;
template<class T, T N>
using make_index_sequence = integer_sequence<T, see below>;
template<class... T>
using index_sequence_for = make_index_sequence<sizeof...(T)>;

// 20.4, class template pair
template<class T1, class T2>
struct pair;

// 20.4.3, pair specialized algorithms
template<class T1, class T2>
constexpr bool operator==(const pair<T1, T2>&, const pair<T1, T2>&);
template<class T1, class T2>
constexpr bool operator!=(const pair<T1, T2>&, const pair<T1, T2>&);
template<class T1, class T2>
constexpr bool operator< (const pair<T1, T2>&, const pair<T1, T2>&);
template<class T1, class T2>
constexpr bool operator> (const pair<T1, T2>&, const pair<T1, T2>&);
template<class T1, class T2>
constexpr bool operator<=(const pair<T1, T2>&, const pair<T1, T2>&);
template<class T1, class T2>
constexpr bool 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<I, pair<T1, T2>>;

template<size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>&) noexcept;
template<size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>&) noexcept;
template<size_t I, class T1, class T2>
    constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>&) noexcept;

template<size_t I, class T1, class T2>
    constexpr const tuple_element_t<I, pair<T1, T2>>&& get(const pair<T1, T2>&&) noexcept;

template<class T1, class T2>
    constexpr T1& get(pair<T1, T2>& p) noexcept;

template<class T1, class T2>
    constexpr const T1& get(const pair<T1, T2>& p) noexcept;

template<class T1, class T2>
    constexpr T1&& get(pair<T1, T2>&& p) noexcept;

template<class T1, class T2>
    constexpr const T1&& get(const pair<T1, T2>&& p) noexcept;

template<class T2, class T1>
    constexpr T2& get(pair<T1, T2>& p) noexcept;

template<class T2, class T1>
    constexpr const T2& get(const pair<T1, T2>& p) noexcept;

template<class T2, class T1>
    constexpr T2&& get(pair<T1, T2>&& p) noexcept;

template<class T2, class T1>
    constexpr const T2&& get(const pair<T1, T2>&& p) noexcept;

// 20.4.5, pair piecewise construction
struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
};

inline constexpr piecewise_construct_t piecewise_construct{};

template<class... Types> class tuple; // defined in <tuple> (20.5.2)

// in-place construction
struct in_place_t {
    explicit in_place_t() = default;
};
inline constexpr in_place_t in_place{};

template<class T> inline constexpr in_place_type_t<T> in_place_type{};

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);

Remarks: This function is a designated customization point (16.5.4.2.1) and shall not participate in
overload resolution unless is_move_constructible_v<T> is true and is_move_assignable_v<T> is true. The expression inside noexcept is equivalent to:

    is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>

Requires: Type T shall be Cpp17MoveConstructible (Table 26) and Cpp17MoveAssignable (Table 28).
Effects: Exchanges values stored in two locations.

template<class T, size_t N>
    constexpr void swap(T (&a)[N], T (&b)[N]) noexcept(is_nothrow_swappable_v<T>);

Remarks: This function shall not participate in overload resolution unless is_swappable_v<T> is true.

Requires: a[i] shall be 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

```cpp
template<class T, class U = T>
constexpr T exchange(T& obj, U&& new_val);
```

**Effects:** Equivalent to:

- `T old_val = std::move(obj);`
- `obj = std::forward<U>(new_val);`
- `return old_val;`

20.2.4 Forward/move helpers

The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions. All functions specified in this subclause are signal-safe (17.13.4).

```cpp
template<class T> constexpr T&& forward(remove_reference_t<T>& t) noexcept;
```

**Returns:** `static_cast<T&&>(t)`.

**Remarks:** If the second form is instantiated with an lvalue reference type, the program is ill-formed.

```cpp
template<class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
  return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}
```

**Example:**

```cpp
struct A {
  A(const A&); // copies from lvalues
  A(A&&); // moves from rvalues
};
```

```cpp
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 `int`, so `a` is forwarded as a non-const lvalue. This binds to the constructor `A(const A&)`, which copies the value from `a`. In the second call to `factory`, because of
ISO/IEC N4810

the call `std::move(a)`, `A1` is deduced as `A`, so `a` is forwarded as an rvalue. This binds to the constructor `A(A&&)`, which moves the value from `a`. —end example

template<class T> constexpr conditional_t<
  !is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>,
  const T& , T&&>
move_if_noexcept(T& x) noexcept;

20.2.5 Function template as_const

template<class T> constexpr add_const_t<T>& as_const(T& t) noexcept;

1 Returns: `t`.

20.2.6 Function template declval

The library provides the function template declval to simplify the definition of expressions which occur as unevaluated operands (7.2).

template<class T> add_rvalue_reference_t<T> declval() noexcept; // as unevaluated operand

1 Remarks: If this function is odr-used (6.2), the program is ill-formed.

2 Remarks: The template parameter `T` of declval may be an incomplete type.

3 [Example:
  template<class To, class From> decltype(static_cast<To>(declval<From>())) convert(From&&);
 declares a function template convert which only participates in overloading if the type `From` can be explicitly converted to type `To`. For another example see class template common_type (20.15.7.6). —end example]

20.3 Compile-time integer sequences

20.3.1 In general

The library provides a class template that can represent an integer sequence. When used as an argument to a function template the template parameter pack defining the sequence can be deduced and used in a pack expansion. [Note: The index_sequence alias template is provided for the common case of an integer sequence of type size_t; see also 20.5.3.5. —end note]

20.3.2 Class template integer_sequence

namespace std {
  template<class T, T... I> struct integer_sequence {
    using value_type = T;
    static constexpr size_t size() noexcept { return sizeof...(I); }
  };
}

1 `T` shall be 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 If `N` is negative the program is ill-formed. The alias template make_integer_sequence denotes a specialization of integer_sequence with `N` template non-type arguments. The type make_integer_sequence<T, N> denotes the type integer_sequence<T, 0, 1, ..., N-1>. [Note: make_integer_sequence<int, 0> denotes the type integer_sequence<int> — end note]

20.4 Pairs

20.4.1 In general

The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to pair objects as if they were tuple objects (see 20.5.3.6 and 20.5.3.7).
20.4.2 Class template pair

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(see below) pair();
    constexpr explicit(see below) pair(const T1& x, const T2& y);
    template<class U1, class U2>
      constexpr explicit(see below) pair(U1&& x, U2&& y);
    template<class U1, class U2>
      constexpr explicit(see below) pair(const pair<U1, U2>& p);
    template<class U1, class U2>
      constexpr explicit(see below) pair(pair<U1, U2>&& p);
    template<class... Args1, class... Args2>
      constexpr pair(piecewise_construct_t,
                      tuple<Args1...> first_args, tuple<Args2...> second_args);
    constexpr pair& operator=(const pair& p);
    template<class U1, class U2>
      constexpr pair& operator=(const pair<U1, U2>& p);
    constexpr pair& operator=(pair&& p) noexcept(see below);
    template<class U1, class U2>
      constexpr pair& operator=(pair<U1, U2>&& p);
    constexpr void swap(pair& p) noexcept(see below);
  }
  template<class T1, class T2>
    pair(T1, T2) -> pair<T1, T2>;
}

1 Constructors and member functions of pair shall 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 shall be a constexpr function if and only if all required element-wise initializations for copy and move, 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 constexpr explicit(see below) pair();

5 Effects: Value-initializes first and second.

6 Remarks: This constructor shall not participate in overload resolution unless is_default_constructible_v<first_type> is true and is_default_constructible_v<second_type> is true. [Note: This behavior can be implemented by a constructor template with default template arguments. — end note] 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]

7 constexpr explicit(see below) pair(const T1& x, const T2& y);

6 Effects: Initializes first with x and second with y.

7 Remarks: This constructor shall not participate in overload resolution unless is_copy_constructible_v<first_type> is true and is_copy_constructible_v<second_type> is true. The expression inside explicit is equivalent to:
template<class U1, class U2> constexpr explicit(see below) pair(U1&& x, U2&& y);

Effects: Initializes first with std::forward<U1>(x) and second with std::forward<U2>(y).

Remarks: This constructor shall not participate in overload resolution unless is_constructible_v<

!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(const pair<U1, U2>& p);

Effects: Initializes members from the corresponding members of the argument.

Remarks: This constructor shall not participate in overload resolution unless is_constructible_v<

!is_convertible_v<const U1&, first_type> || !is_convertible_v<const U2&, second_type>

template<class... Args1, class... Args2>
constexpr pair(piecewise_construct_t, tuple<Args1...> first_args, tuple<Args2...> second_args);

Requires: is_constructible_v<first_type, Args1&&...> is true and 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 shall be defined as deleted unless is_copy_assignable_v<first_type> is true and is_copy_assignable_v<second_type> is true.

Returns: *this.

template<class U1, class U2> constexpr pair& operator=(const pair<U1, U2>& p);

Effects: Assigns p.first to first and p.second to second.

Remarks: This operator shall participate in overload resolution unless is_assignable_v<

!is_convertible_v<const U1&, first_type> || !is_convertible_v<const U2&, second_type>

constexpr pair& operator=(pair&& p) noexcept(see below);

Effects: Assigns to first with std::forward<first_type>(p.first) and to second with std::forward<second_type>(p.second).

Remarks: This operator shall not participate in overload resolution unless is_move_assignable_v<

!is_convertible_v<first_type> || !is_convertible_v<second_type>

§ 20.4.2 550
is_nothrow_move_assignable_v<T1> && is_nothrow_move_assignable_v<T2>

Returns: *this.

template<class U1, class U2>
constexpr pair& operator=(pair<U1, U2>&& p);

Effects: Assigns to first with \texttt{std::forward<U1>(p.\texttt{first})} and to second with \texttt{std::forward<U2>(p.\texttt{second})}.

Remarks: This operator shall not participate in overload resolution unless \texttt{is assignable_v<first_type, U1\&\&> is true and is assignable_v<second_type, U2\&\&> is true.}

Returns: *this.

constexpr void swap(pair& p) noexcept;

Requires: first shall be swappable with (16.5.3.2) p.first and second shall be swappable with p.second.

Effects: Swaps first with p.first and second with p.second.

Remarks: The expression inside noexcept is equivalent to:

\texttt{is_{nothrow\_swappable\_v<first\_type> \&\& is_{nothrow\_swappable\_v<second\_type>}}

\subsection*{20.4.3 Specialized algorithms} [pairs.spec]

\begin{verbatim}
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 bool operator!=(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: !(x == y).

template<class T1, class T2>
constexpr bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: x.first < y.first || (!y.first < x.first) \&\& x.second < y.second).

template<class T1, class T2>
constexpr bool operator<=(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: !(y < x).

template<class T1, class T2>
constexpr bool operator>(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: !(x < y).

template<class T1, class T2>
constexpr bool operator>=(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: !(x < y).

template<class T1, class T2>
constexpr void swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y)));

Effects: As if by x.swap(y).

Remarks: This function shall not participate in overload resolution unless \texttt{is\_swappable\_v<T1> is true and is\_swappable\_v<T2> is true.}

template<class T1, class T2>
constexpr pair<unwrap_ref_decay_t<T1>, unwrap_ref_decay_t<T2>> make_pair(T1&& x, T2&& y);

Returns:

\begin{verbatim}
pair<unwrap_ref_decay_t<T1>,
  unwrap_ref_decay_t<T2>>(
  std::forward<T1>(x),
  std::forward<T2>(y))
\end{verbatim}

[Example: In place of:

\begin{verbatim}
return pair<int, double>(5, 3.1415926);
\end{verbatim}

// explicit types
\end{verbatim}
\end{verbatim}
\end{verbatim}

\section*{§ 20.4.3 551}
a C++ program may contain:

```cpp
return make_pair(5, 3.1415926);  // types are deduced
```

— end example]

### 20.4.4 Tuple-like access to pair

```cpp
template<class T1, class T2>
struct tuple_size<pair<T1, T2>> : integral_constant<size_t, 2> { };

tuple_element<I, pair<T1, T2>>::type
```  

1. **Requires**: $I < 2$. The program is ill-formed if $I$ is out of bounds.

2. **Value**: The type $T_1$ if $I == 0$, otherwise the type $T_2$.

```cpp
template<size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>& p) noexcept;
template<size_t I, class T1, class T2>
constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>& p) noexcept;
template<size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>& get(pair<T1, T2>&& p) noexcept;
template<size_t I, class T1, class T2>
constexpr const tuple_element_t<I, pair<T1, T2>>& get(const pair<T1, T2>&& p) noexcept;
```

3. **Returns**: If $I == 0$ returns a reference to $p$.first; if $I == 1$ returns a reference to $p$.second; otherwise the program is ill-formed.

```cpp
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;
```

4. **Requires**: $T_1$ and $T_2$ are distinct types. Otherwise, the program is ill-formed.

```cpp
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;
```

5. **Requires**: $T_1$ and $T_2$ are distinct types. Otherwise, the program is ill-formed.

### 20.4.5 Piecewise construction

```cpp
struct piecewise_construct_t {
    explicit piecewise_construct_t() = default;
};
inline constexpr piecewise_construct_t piecewise_construct{};
```

1. The struct `piecewise_construct_t` is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, `pair` has a constructor with `piecewise_construct_t` as the first argument, immediately followed by two `tuple` (20.5) arguments used for piecewise construction of the elements of the `pair` object.
20.5 Tuples

20.5.1 In general

This subclause 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

```cpp
namespace std {
  // 20.5.3, class template tuple
template<class... Types>
  class tuple;

  // 20.5.3.4, 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<Tuples&&...> tuple_cat(Tuples&&...);

  // 20.5.3.5, 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);

  // 20.5.3.6, tuple helper classes
  template<class T> struct tuple_size; // not defined
  template<class T> struct tuple_size<const T>;
template<class T> struct tuple_size<volatile T>;
template<class T> struct tuple_size<const volatile T>;
template<class... Types> struct tuple_size<tuple<Types...>>;
template<size_t I, class T> struct tuple_element; // not defined
template<size_t I, class T> struct tuple_element<I, const T>;
template<size_t I, class T> struct tuple_element<I, volatile T>;
template<size_t I, class T> struct tuple_element<I, const volatile T>;
template<size_t I, class... Types>
  struct tuple_element<I, tuple<Types...>>;
template<size_t I, class T>
  using tuple_element_t = typename tuple_element<I, T>::type;

  // 20.5.3.7, element access
  template<size_t I, class... Types>
    constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&) noexcept;
template<size_t I, class... Types>
    constexpr tuple_element_t<I, tuple<Types...>>&& get(tuple<Types...>&&) noexcept;
template<size_t I, class... Types>
    constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>&) noexcept;
```

§ 20.5.2 553
template<
    size_t I,
    class... Types>
constexpr
const tuple_element_t<I, tuple<Types...>>
&& get(const tuple<Types...>&& t) noexcept;

template<class T, class... Types>
constexpr
T & get(tuple<Types...>& t) noexcept;

template<class T, class... Types>
constexpr
T && get(tuple<Types...>&& t) noexcept;

template<class T, class... Types>
constexpr
const T & get(const tuple<Types...>& t) noexcept;

template<class T, class... Types>
constexpr
const T && get(const tuple<Types...>&& t) noexcept;

// 20.5.3.8, relational operators

template<class... TTypes, class... UTypes>
constexpr
bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class... TTypes, class... UTypes>
constexpr
bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class... TTypes, class... UTypes>
constexpr
bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class... TTypes, class... UTypes>
constexpr
bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class... TTypes, class... UTypes>
constexpr
bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);

template<class... TTypes, class... UTypes>
constexpr
bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

// 20.5.3.9, allocator-related traits

template<class... Types, class Alloc>
struct
uses_allocator<tuple<Types...>, Alloc>;

// 20.5.3.10, specialized algorithms

template<class... Types>
constexpr
void
swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);

// 20.5.3.6, tuple helper classes

template<class T>
inline
costexpr
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

}}

§ 20.5.3

554
// allocator-extended constructors
template<class Alloc>
constexpr tuple(allocator_arg_t, const Alloc& a);

template<class Alloc>
constexpr explicit(see below)
    tuple(allocator_arg_t, const Alloc& a, const Types&...);

// only if sizeof...(Types) == 2
template<class Alloc, class U1, class U2>
tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&); // see below

// 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 Alloc, class... UTypes>
tuple<allocator_arg_t, Alloc, tuple<UTypes...>> -> tuple<UTypes...>;

e 20.5.3.3, tuple swap
constexpr void swap(tuple&) noexcept(see below);

20.5.3.1 Construction [tuple.cnstr]
1 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.
2 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` shall be a constexpr function if and only if all required element-wise initializations for copy and move, respectively, would satisfy the requirements for a constexpr function. The defaulted move and copy constructor of `tuple<>` shall be constexpr functions.

If `is_trivially_deductible_v<T_i>` is true for all `T_i`, then the destructor of `tuple` is trivial.

```cpp
constexpr explicit(see below) tuple();
```

**Effects:** Value-initializes each element.

**Remarks:** This constructor shall not participate in overload resolution unless `is_default_constructible_v<T_i>` is true for all `i`. [Note: This behavior can be implemented by a constructor template with default template arguments. —end note] The expression inside `explicit` evaluates to **true** if and only if `T_i` is not implicitly default-constructible for at least one `i`. [Note: This behavior can be implemented with a trait that checks whether a `const T_i&` can be initialized with `{}`. —end note]

```cpp
constexpr explicit(see below) tuple(const Types&...);
```

**Effects:** Initializes each element with the value of the corresponding parameter.

**Remarks:** This constructor shall not participate in overload resolution unless `sizeof...(Types) >= 1` and `is_copy_constructible_v<T_i>` is true for all `i`. The expression inside `explicit` is equivalent to:

```cpp
!conjunction_v<is_convertible<const Types&, Types>...>
```

### § 20.5.3.1 556
either sizeof...(Types) != 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.

The expression inside explicit is equivalent to:

!conjunction_v<is_convertible<UTypes, Types>...>

This constructor shall not participate in overload resolution unless sizeof...(Types) == 2, is_constructible_v<T0, const U1&> is true and is_constructible_v<T1, const U2&> is true.

The expression inside explicit is equivalent to:

!is_convertible_v<const U1&, T0> || !is_convertible_v<const U2&, T1>

This constructor shall not participate in overload resolution unless sizeof...(Types) == 2, is_constructible_v<T0, U1&&> is true and is_constructible_v<T1, U2&&> is true.

The expression inside explicit is equivalent to:

!is_convertible_v<U1, T0> || !is_convertible_v<U2, T1>

Requires: Alloc shall satisfy the Cpp17Allocator requirements (Table 34). Effects: Equivalent to the preceding constructors except that each element is constructed with uses-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.

constexpr tuple& operator=(const tuple& u);

Effects: Assigns each element of u to the corresponding element of *this.
Remarks: This operator shall be defined as deleted unless \texttt{is\_copy\_assignable\_v<T\_i>} is true for all \texttt{i}.

Returns: \texttt{*this}.

```cpp
constexpr tuple& operator=(tuple&& u) noexcept(see below);
```

Effects: For all \texttt{i}, assigns \texttt{std::forward<T\_i>\(>(\texttt{get<i>}(\texttt{u}))\) to \texttt{get<i>\((\texttt{this})\).}

Remarks: This operator shall not participate in overload resolution unless \texttt{is\_move\_assignable\_v<T\_i>} is true for all \texttt{i}.

Remarks: The expression inside \texttt{noexcept} is equivalent to the logical AND of the following expressions:

\texttt{is\_nothrow\_move\_assignable\_v<T\_i>}

where \texttt{T\_i} is the \texttt{i}th type in \texttt{Types}.

Returns: \texttt{*this}.

```cpp
template<class... UTypes> constexpr tuple& operator=(const tuple<UTypes...>& u);
```

Effects: Assigns each element of \texttt{u} to the corresponding element of \texttt{*this}.

Remarks: This operator shall not participate in overload resolution unless \texttt{sizeof\(\ldots\)(Types) == sizeof\(\ldots\)(UTypes)} and \texttt{is\_assignable\_v<T\_i&, const U\_i&> is true for all \texttt{i}}.

Returns: \texttt{*this}.

```cpp
template<class... UTypes> constexpr tuple& operator=(tuple<UTypes...>&& u);
```

Effects: For all \texttt{i}, assigns \texttt{std::forward<U\_i>(\texttt{get<i>}(\texttt{u}))\) to \texttt{get<i>\((\texttt{this})\).}

Remarks: This operator shall not participate in overload resolution unless \texttt{is\_assignable\_v<T\_i&, U\_i&&> == true for all \texttt{i}} and \texttt{sizeof\(\ldots\)(Types) == sizeof\(\ldots\)(UTypes)}.

Returns: \texttt{*this}.

```cpp
template<class U1, class U2> constexpr tuple& operator=(const pair<U1, U2>& u);
```

Effects: Assigns \texttt{u.first} to the first element of \texttt{*this} and \texttt{u.second} to the second element of \texttt{*this}.

Remarks: This operator shall not participate in overload resolution unless \texttt{sizeof\(\ldots\)(Types) == 2} and \texttt{is\_assignable\_v<T\_0&, const U1&> is true for the first type \texttt{T\_0} in \texttt{Types} and \texttt{is\_assignable\_v<T\_1&, const U2&> is true for the second type \texttt{T\_1 in Types}}.

Returns: \texttt{*this}.

```cpp
template<class U1, class U2> constexpr tuple& operator=(pair<U1, U2>&& u);
```

Effects: Assigns \texttt{std::forward<U1>(u.first)} to the first element of \texttt{*this} and \texttt{std::forward<U2>(u.second)} to the second element of \texttt{*this}.

Remarks: This operator shall not participate in overload resolution unless \texttt{sizeof\(\ldots\)(Types) == 2} and \texttt{is\_assignable\_v<T\_0&, U1&&> is true for the first type \texttt{T\_0} in \texttt{Types} and \texttt{is\_assignable\_v<T\_1&, U2&&> is true for the second type \texttt{T\_1 in Types}}.

Returns: \texttt{*this}.

### 20.5.3.3 swap [tuple.swap]

```cpp
constexpr void swap(tuple& rhs) noexcept(see below);
```

Requires: Each element in \texttt{*this} shall be swappable with (16.5.3.2) the corresponding element in \texttt{rhs}.

Effects: Calls \texttt{swap} for each element in \texttt{*this} and its corresponding element in \texttt{rhs}.

Remarks: The expression inside \texttt{noexcept} is equivalent to the logical AND of the following expressions:

\texttt{is\_nothrow\_swappable\_v<T\_i>}

where \texttt{T\_i} is the \texttt{i}th type in \texttt{Types}.

Throws: Nothing unless one of the element-wise \texttt{swap} calls throws an exception.

### 20.5.3.4 Tuple creation functions [tuple.creation]

In the function descriptions that follow, the members of a template parameter pack \texttt{XTypes} are denoted by \texttt{X\_i} for \texttt{i} in \{0, \texttt{sizeof\(\ldots\)(XTypes)}\} in order, where indexing is zero-based.
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 variables, 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>
tuple_cat(Tuples&&... tpls);

In the following paragraphs, let T_i be the i^{th} type in Tuples, U_i be remove_reference_t<T_i>, and tp_i be the i^{th} parameter in the function parameter pack tpls, where all indexing is zero-based.

Requires: For all i, U_i shall be the type cv_i tuple<Args_i>..., where cv_i is the (possibly empty) i^{th} cv-qualifier-seq and Args_i is the template parameter pack representing the element types in U_i. Let A_i_k be the k^{th} type in Args_i. For all A_i_k the following requirements shall be satisfied:

- If T_i is deduced as an lvalue reference type, then is_constructible_v<A_i_k, cv_i A_i_k&> == true, otherwise
- is_constructible_v<A_i_k, cv_i A_i_k&&> == true.

Remarks: The types in CTypes shall be equal to the ordered sequence of the extended types Args_0..., Args_1..., ..., Args_{n-1}..., where n is equal to sizeof...(Tuples). Let e_i... be the i^{th} ordered sequence of tuple elements of the resulting tuple object corresponding to the type sequence Args_i.

Returns: A tuple object constructed by initializing the k_i^{th} type element e_i_k in e_i... with
get<k_i>(std::forward<T_i>(tp_i))
for each valid k_i and each group e_i in order.

[Note: An implementation may support additional types in the template parameter pack Tuples that support the tuple-like protocol, such as pair and array. — end note]

20.5.3.5 Calling a function with a tuple of arguments [tuple.apply]

template<class F, class Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t);

Effects: Given the exposition-only function:

template<class F, class Tuple, size_t... I>
constexpr decltype(auto) apply-impl(F&& f, Tuple&& t, index_sequence<I...>) {
  // exposition only

§ 20.5.3.5 559
return \textit{INVOK}e(std::forward\(<\text{F}\>\(\text{f}\), std::get\(<\text{I}\>(std::forward\(<\text{Tuple}\>\(\text{t}\)))\...);  \hspace{3em} // \text{see \ref{std::in Victoria}}
}

Equivalent to:

\begin{verbatim}
return apply-impl(std::forward<\text{F}\>(\text{f}), std::forward<\text{Tuple}\>(\text{t}),
make_index_sequence<\text{tuple_size}_v<\text{remove_reference}\_t<\text{Tuple}>\{}\{});
\end{verbatim}

\begin{verbatim}
template<class \text{T}, class \text{Tuple}>
constexpr \text{T} make_from_tuple(Tuple&& \text{t});
\end{verbatim}

\textbf{Effects:} Given the exposition-only function:

\begin{verbatim}
template<class \text{T}, class \text{Tuple}, size_t... \text{I}>
constexpr \text{T} make-from-tuple-impl(Tuple&& \text{t}, index_sequence<\text{I}...>) {  \hspace{3em} // exposition only
return \text{T}(get<\text{I}>(std::forward<\text{Tuple}\>(\text{t}))...);
}
\end{verbatim}

Equivalent to:

\begin{verbatim}
return make-from-tuple-impl<\text{T}\>(
forward<\text{Tuple}\>(\text{t}),
make_index_sequence<\text{tuple_size}_v<\text{remove_reference}\_t<\text{Tuple}>\{}\{});
\end{verbatim}

\textbf{[Note: The type of \text{T} must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. — end note]}

\section*{20.5.3.6 Tuple helper classes}[tuple.helper]

\begin{verbatim}
template<class \text{T}>
struct tuple_size;
\end{verbatim}

\textbf{Remarks:} All specializations of \text{tuple_size} shall satisfy the \textit{Cpp17UnaryTypeTrait} requirements (20.15.1) with a base characteristic of \textit{integral_constant}<\text{size_t}, \text{N}> for some \text{N}.

\begin{verbatim}
template<class... \text{Types}>
struct tuple_size<tuple<\text{Types}...>> : public integral_constant<\text{size_t}, sizeof...(\text{Types})> {  \{};
\end{verbatim}

\begin{verbatim}
template<size_t \text{I}, class... \text{Types}>
struct tuple_element<\text{I}, tuple<\text{Types}...>> {  \{}
using type = \text{TI};
\};
\end{verbatim}

\textbf{Requires:} \text{I} < \text{sizeof...(\text{Types})}. The program is ill-formed if \text{I} is out of bounds.

\textbf{Type:} \text{TI} is the type of the \text{I}^{th} element of \text{Types}, where indexing is zero-based.

\begin{verbatim}
template<class \text{T}>
struct tuple_size<const \text{T}>;
template<class \text{T}>
struct tuple_size<volatile \text{T}>;
template<class \text{T}>
struct tuple_size<const volatile \text{T}>;
\end{verbatim}

\textbf{Let TS denote tuple_size<\text{T}> of the cv-unqualified type \text{T}. If the expression TS::\text{value} is well-formed when treated as an unevaluated operand, then each of the three templates shall satisfy the Cpp17UnaryTypeTrait requirements (20.15.1) with a base characteristic of integral_constant<\text{size_t}, TS::\text{value}>.

Otherwise, they shall have no member \text{value}.

Access checking is performed as if in a context unrelated to TS and \text{T}. 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]

In addition to being available via inclusion of the \textless \text{tuple} \textgreater header, the three templates are available when any of the headers \textless \text{array} \textgreater, \textless \text{ranges} \textgreater, \textless \text{span} \textgreater, or \textless \text{utility} \textgreater are included.

\begin{verbatim}
template<size_t \text{I}, class \text{T}>
struct tuple_element<\text{I}, const \text{T}>;
template<size_t \text{I}, class \text{T}>
struct tuple_element<\text{I}, volatile \text{T}>;
\end{verbatim}
template<
    size_t I, class T>
struct tuple_element<I, const volatile T>;

Let TE denote tuple_element_t<I, T> of the cv-unqualified type T. Then each of the three templates shall satisfy the Cpp17TransformationTrait requirements (20.15.1) with a member typedef type that names the following type:

(7.1) for the first specialization, add_const_t<TE>,
(7.2) for the second specialization, add_volatile_t<TE>, and
(7.3) for the third specialization, add_cv_t<TE>.

In addition to being available via inclusion of the <tuple> header, the three templates are available when any of the headers <array>, <ranges>, <span>, or <utility> are included.

20.5.3.7 Element access

template<
    size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>> &
get(tuple<Types...>& t) noexcept;

template<
    size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&&
get(tuple<Types...>&& t) noexcept; // Note A

template<
    size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>> &
get(const tuple<Types...>& t) noexcept; // Note B

template<
    size_t I, class... Types>
constexpr const tuple_element_t<I, tuple<Types...>>&&
get(const tuple<Types...>&& t) noexcept;

Requires: I < sizeof...(Types). The program is ill-formed if I is out of bounds.

Returns: A reference to the I\textsuperscript{th} element of \( t \), where indexing is zero-based.

[Note A: If a type \( T \) in \( \text{Types} \) is some reference type \( X\& \), the return type is \( X\& \), not \( X&& \). However, if the element type is a non-reference type \( T \), the return type is \( T&& \). —end note]

[Note B: Constness is shallow. If a type \( T \) in \( \text{Types} \) is some reference type \( X\& \), the return type is \( X\& \), not \( \text{const } X\& \). However, if the element type is a non-reference type \( T \), the return type is \( \text{const } T\& \). This is consistent with how constness is defined to work for member variables of reference type. —end note]

template<
    class T, class... Types>
constexpr T&
get(tuple<Types...>& t) noexcept;

template<
    class T, class... Types>
constexpr T&&
get(tuple<Types...>&& t) noexcept;

template<
    class T, class... Types>
constexpr const T&
get(const tuple<Types...>& t) noexcept;

template<
    class T, class... Types>
constexpr const T&&
get(const tuple<Types...>&& t) noexcept;

Requires: The type \( T \) occurs exactly once in \( \text{Types} \). Otherwise, the program is ill-formed.

Returns: A reference to the element of \( t \) corresponding to the type \( T \) in \( \text{Types} \).

[Example:

\begin{verbatim}
const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
const int& i1 = get<int>(t); // OK. Not ambiguous. i1 == 1
const int& i2 = get<const int>(t); // OK. Not ambiguous. i2 == 2
const double& d = get<double>(t); // ERROR. ill-formed
\end{verbatim}

— 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 \texttt{template} keyword. —end note]

20.5.3.8 Relational operators

template<
    class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>&& t, const tuple<UTypes...>&& u);

Requires: For all \( i \), where \( 0 \leq i < \text{sizeof}...(TTypes) \), get\(i\)(t) == get\(i\)(u) is a valid expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).

§ 20.5.3.8
Returns: `true` if \( \text{get}<i>(t) == \text{get}<i>(u) \) 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`.

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: \(! (t == u)\).

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Requires: For all \( i \), where \( 0 <= i \) and \( i < \text{sizeof}(\ldots)(\text{TTypes}) \), both \( \text{get}<i>(t) < \text{get}<i>(u) \) and \( \text{get}<i>(u) < \text{get}<i>(t) \) are valid expressions returning types that are convertible to `bool`. \( \text{sizeof}(\ldots)(\text{TTypes}) == \text{sizeof}(\ldots)(\text{UTypes}) \).

Returns: The result of a lexicographical comparison between \( t \) and \( u \). The result is defined as: 
\[
(\text{bool})(\text{get}<0>(t) < \text{get}<0>(u)) \ || \ (! (\text{bool})(\text{get}<0>(u) < \text{get}<0>(t)) && \text{t}_\text{tail} < \text{u}_\text{tail}),
\]
where \( r_{\text{tail}} \) for some tuple \( r \) is a tuple containing all but the first element of \( r \). For any two zero-length tuples \( e \) and \( f \), \( e < f \) returns `false`.

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: \( u < t \).

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: \(! (u < t)\).

```cpp
template<class... TTypes, class... UTypes>
constexpr bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

Returns: \(! (t < u)\).

[Note: The above definitions for comparison functions do not require \( t_{\text{tail}} \) (or \( u_{\text{tail}} \)) 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]

20.5.3.9 Tuple traits

```cpp
template<class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc> : true_type { }; 
```

Requires: Alloc shall satisfy the Cpp17Allocator requirements (Table 34).

[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]

20.5.3.10 Tuple specialized algorithms

```cpp
template<class... Types>
constexpr void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below); 
```

Remarks: This function shall not participate in overload resolution unless \( \text{is\_swappable}\_\text{v}<\text{T}> \) is \( \text{true} \) for every type \( T \) in \( \text{Types} \). The expression inside noexcept is equivalent to:

\[ \text{noexcept(x.swap(y))} \]

Effects: As if by \( x\_\text{swap}(y) \).

20.6 Optional objects

20.6.1 In general

This subclause describes class template \text{optional} that represents optional objects. An \text{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

```cpp
namespace std {
    // 20.6.3, class template optional
    template<class T>
    class optional;

    // 20.6.4, no-value state indicator
    struct nullopt_t{see below};
    inline constexpr nullopt_t nullopt_(unspecified);

    // 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>&);

    // 20.6.7, comparison with nullopt
    template<class T> constexpr bool operator==(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr bool operator==(nullopt_t, const optional<T>&) noexcept;
    template<class T> constexpr bool operator!=(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr bool operator!=(nullopt_t, const optional<T>&) noexcept;
    template<class T> constexpr bool operator<(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr bool operator<(nullopt_t, const optional<T>&) noexcept;
    template<class T> constexpr bool operator>(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr bool operator>(nullopt_t, const optional<T>&) noexcept;
    template<class T> constexpr bool operator<=(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr bool operator<=(nullopt_t, const optional<T>&) noexcept;
    template<class T> constexpr bool operator>=(const optional<T>&, nullopt_t) noexcept;
    template<class T> constexpr bool operator>=(nullopt_t, const optional<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>&);

    // 20.6.9, specialized algorithms
    template<class T>
    void swap(optional<T>&, optional<T>&) noexcept(see below);
```
template<class T>
constexpr optional<see below> make_optional(T&&);

// 20.6.10, hash support
// 3.11.1.1, hash functions

namespace std {
    template<class T> struct optional {
        using value_type = T;

        // 20.6.3.1, constructors
        constexpr optional() noexcept;
        constexpr optional(nullopt_t) noexcept;
        constexpr optional(const optional&);
        template<class... Args>
        constexpr optional(Args&&...);
        template<class U, class... Args>
        constexpr optional(initializer_list<U> il, Args&&... args);

        // 20.6.3.2, destructor
        ~optional();

        // 20.6.3.3, assignment
        optional& operator=(nullopt_t) noexcept;
        constexpr optional& operator=(const optional&);
        constexpr optional& operator=(optional&&) noexcept;
        template<class U = T>
        optional& operator=(U&&);
        template<class U> 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;

        // 20.6.3.5, observers
        constexpr const T* operator->() const;
        constexpr T* operator->();
        constexpr const T& operator*() const&;
        constexpr T& operator*() &;
        constexpr T&& operator*() &&;
        constexpr const T&& operator*() const&&;
        constexpr explicit operator bool() const noexcept;
        constexpr bool has_value() const noexcept;
        constexpr const T& value() const&;
        constexpr T& value() &;
        constexpr T&& value() &&;
    };
    template<class T, class... Args>
    constexpr optional<T> make_optional(Args&&... args);
    template<class T, class U, class... Args>
    constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);
}

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&);
            template<class... Args>
            constexpr explicit optional(in_place_t, Args&&...);
            template<class U, class... Args>
            constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
            template<class U = T>
            explicit(see below) constexpr optional(U&&);
            template<class U>
            explicit(see below) optional(const optional<U>&);
            template<class U = T>
            explicit(see below) constexpr optional(U&&);

            // 20.6.3.2, destructor
            ~optional();

            // 20.6.3.3, assignment
            optional& operator=(nullopt_t) noexcept;
            constexpr optional& operator=(const optional&);
            constexpr optional& operator=(optional&&) noexcept;
            template<class U = T>
            optional& operator=(U&&);
            template<class U> optional& operator=(const optional<U>&);
            template<class U> optional& operator=(optional<U>&&);
            template<class... Args> T& emplace(Args&&...);

            // 20.6.3.4, swap
            void swap(optional&) noexcept;

            // 20.6.3.5, observers
            constexpr const T* operator->() const;
            constexpr T* operator->();
            constexpr const T& operator*() const&;
            constexpr T& operator*() &;
            constexpr T&& operator*() &&;
            constexpr const T&& operator*() const&&;
            constexpr explicit operator bool() const noexcept;
            constexpr bool has_value() const noexcept;
            constexpr const T& value() const&;
            constexpr T& value() &;
            constexpr T&& value() &&;
    };
}
constexpr const T&& value() const&&;

template<class U> constexpr T value_or(U&&) const&;

// 20.6.3.6, modifiers
void reset() noexcept;

private:
    T *val;  // exposition only
};

template<class T>
optional(T) -> optional<T>;

Any instance of `optional<T>` at any given time either contains a value or does not contain a value. When
an instance of `optional<T>` contains a value, it means that an object of type `T`, referred to as the optional
object’s contained value, is allocated within the storage of the optional object. Implementations are not
permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained
value shall be allocated in a region of the `optional<T>` storage suitably aligned for the type `T`. When an
object of type `optional<T>` is contextually converted to `bool`, the conversion returns `true` if the object
contains a value; otherwise the conversion returns `false`.

Member `val` is provided for exposition only. When an `optional<T>` object contains a value, `val` points to
the contained value.

`T` shall be an object type other than `cv in_place_t` or `cv nullopt_t` and shall satisfy the `Cpp17Destructible`
requirements (Table 30).

### 20.6.3.1 Constructors

```cpp
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;

1 Ensures: *this does not contain a value.
2 Remarks: No contained value is initialized. For every object type `T` these constructors shall be constexpr
constructors (9.1.5).
```

```cpp
constexpr optional(const optional& rhs);
```

3 Effects: If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object
of type `T` with the expression `*rhs`.

4 Ensures: `bool(rhs) == bool(*this)`.
5 Throws: Any exception thrown by the selected constructor of `T`.

6 Remarks: This constructor shall be defined as deleted unless `is_copy_constructible_v<T>` is true.
If `is_trivially_copy_constructible_v<T>` is true, this constructor is trivial.

```cpp
constexpr optional(optional&& rhs) noexcept;
```

7 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.

8 Ensures: `bool(rhs) == bool(*this)`.
9 Throws: Any exception thrown by the selected constructor of `T`.

10 Remarks: The expression inside `noexcept` is equivalent to `is_nothrow_move_constructible_v<T>`.
This constructor shall not participate in overload resolution unless `is_move_constructible_v<T>` is
true. If `is_trivially_move_constructible_v<T>` is true, this constructor is trivial.

```cpp
template<class... Args> constexpr explicit optional(in_place_t, Args&&... args);
```

11 Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments
`std::forward<Args>(args)`.

12 Ensures: `*this` contains a value.
13 Throws: Any exception thrown by the selected constructor of `T`.

§ 20.6.3.1
Remarks: If T's constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor. This constructor shall not participate in overload resolution unless is_constructible_v<T, Args...> is true.

```cpp
template<class U, class... Args>
constexpr explicit optional(in_place_t, initializer_list<U> il, Args&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Args>(args)....

Ensures: *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 shall be a constexpr constructor.

```cpp
template<class U, class... Args>
constexpr explicit optional(in_place_t, initializer_list<U> il, Args&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Args>(args)....

Ensures: *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 shall be a constexpr constructor.

```cpp
template<class U = T> explicit(U&& v);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the expression std::forward<U>(v).

Ensures: *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 shall be a constexpr constructor. This constructor shall not participate in overload resolution unless 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<T>, optional> is false. The expression inside explicit is equivalent to:

!is_convertible_v<U, T>

```cpp
template<class U> explicit(see below) constexpr optional(U&& v);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the expression std::forward<U>(v).

Ensures: *this contains a value.

Throws: Any exception thrown by the selected constructor of T.

Remarks: This constructor shall not participate in overload resolution unless

1. is_constructible_v<T, const U&> is true,
2. is_constructible_v<T, optional<U>&> is false,
3. is_constructible_v<T, optional<U>&&> is false,
4. is_constructible_v<T, const optional<U>&> is false,
5. is_constructible_v<T, const optional<U>&&> is false,
6. is_convertible_v<optional<U>&, T> is false,
7. is_convertible_v<optional<U>&&, T> is false,
8. is_convertible_v<const optional<U>&, T> is false, and
9. is_convertible_v<const optional<U>&&, T> is false.

The expression inside explicit is equivalent to:

!is_convertible_v<const U, T>

```cpp
template<class U> explicit(see below) optional(optional<U>& 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 std::move(*rhs). bool(rhs) is unchanged.

Ensures: bool(rhs) == bool(*this).

Throws: Any exception thrown by the selected constructor of T.

Remarks: This constructor shall not participate in overload resolution unless

1. 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<const optional<U>&, T> is false,
— is_convertible_v<const optional<U>&&, T> is false.

The expression inside explicit is equivalent to:

!is_convertible_v<U, T>

20.6.3.2 Destructor

~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<T>& operator=(nullopt_t) noexcept;

Effects: If *this contains a value, calls val->T::~T() to destroy the contained value; otherwise no effect.

Returns: *this.

Ensures: *this does not contain a value.

constexpr optional<T>& operator=(const optional& rhs);

Effects: See Table 40.

Table 40 — optional::operator=(const optional&) effects

<table>
<thead>
<tr>
<th>rhs contains a value</th>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>assigns *rhs to the contained value</td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with *rhs</td>
<td></td>
</tr>
</tbody>
</table>

| rhs does not contain a value | destroys the contained value by calling val->T::~T() | no effect |

Returns: *this.

Ensures: bool(rhs) == bool(*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 shall be defined as deleted unless is_copy_constructible_v<T> is true and is_copy_assignable_v<T> is true. If is_trivially_copy_constructible_v<T> && is_trivially_copy_assignable_v<T> && is_trivially_destructible_v<T> is true, this assignment operator is trivial.

constexpr optional& operator=(optional&& rhs) noexcept(see below);

Effects: See Table 41. The result of the expression bool(rhs) remains unchanged.

Returns: *this.

Ensures: bool(rhs) == bool(*this).
Table 41 — `optional::operator=(optional&&)` effects

<table>
<thead>
<tr>
<th><strong>rhs contains a value</strong></th>
<th>*<strong>this contains a value</strong></th>
<th>*<strong>this does not contain a value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>assigns <code>std::move(*rhs)</code> to the contained value</td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with <code>std::move(*rhs)</code></td>
<td></td>
</tr>
<tr>
<td><strong>rhs does not contain a value</strong></td>
<td>destroys the contained value by calling <code>val-&gt;T::~T()</code></td>
<td>no effect</td>
</tr>
</tbody>
</table>

**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. This operator shall not participate in overload resolution unless `is_move_constructible_v<T>` is true and `is_move_assignable_v<T>` is true. 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.

```cpp
template<class U = T> optional<T>& operator=(U&& v);
```

**Effects:** If `*this` contains a value, assigns `std::forward<U>(v)` to the contained value; otherwise initializes the contained value as if direct-non-list-initializing object of type T with `std::forward<U>(v)`.

**Returns:** `*this`.

**Ensures:** `*this` contains a value.

**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 v 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 v is determined by the exception safety guarantee of T's assignment. This function shall not participate in overload resolution unless `is_same_v<remove_cvref_t<U>, optional>` is false, `conjunction_v<is_scalar<T>, is_same<T, decay_t<U>>>` is false, `is_constructible_v<T, U>` is true, and `is_assignable_v<T&, U>` is true.

```cpp
template<class U> optional<T>& operator=(const optional<U>& rhs);
```

**Effects:** See Table 42.

Table 42 — `optional::operator=(const optional<U>&)` effects

<table>
<thead>
<tr>
<th><strong>rhs contains a value</strong></th>
<th>*<strong>this contains a value</strong></th>
<th>*<strong>this does not contain a value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>assigns <code>*rhs</code> to the contained value</td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with <code>*rhs</code></td>
<td></td>
</tr>
<tr>
<td><strong>rhs does not contain a value</strong></td>
<td>destroys the contained value by calling <code>val-&gt;T::~T()</code></td>
<td>no effect</td>
</tr>
</tbody>
</table>

**Returns:** `*this`.

**Ensures:** `bool(rhs) == bool(*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 constructor, the state of `*rhs.val` is determined by the exception safety guarantee of T's constructor.
assignment, the state of \*val and \*rhs.val is determined by the exception safety guarantee of T's assignment. This function shall not participate in overload resolution unless

(20.1) \(\text{is\_constructible\_v<T, const U&> is true,}\)
(20.2) \(\text{is\_assignable\_v<T&, const U&> is true,}\)
(20.3) \(\text{is\_constructible\_v<T, optional<U>&> is false,}\)
(20.4) \(\text{is\_constructible\_v<T, optional<U>&&> is false,}\)
(20.5) \(\text{is\_constructible\_v<T, const optional<U>&> is false,}\)
(20.6) \(\text{is\_constructible\_v<T, const optional<U>&&> is false,}\)
(20.7) \(\text{is\_convertible\_v<optional<U>&, T> is false,}\)
(20.8) \(\text{is\_convertible\_v<optional<U>&&}, T> is false,}\)
(20.9) \(\text{is\_convertible\_v<const optional<U>&, T> is false,}\)
(20.10) \(\text{is\_convertible\_v<const optional<U>&&}, T> is false,}\)
(20.11) \(\text{is\_assignable\_v<T&, optional<U>&> is false,}\)
(20.12) \(\text{is\_assignable\_v<T&, optional<U>&&> is false,}\)
(20.13) \(\text{is\_assignable\_v<T&, const optional<U>&> is false,}\)
(20.14) \(\text{is\_assignable\_v<T&, const optional<U>&&> is false.}\)

```cpp
template<class U> optional<T>& operator=(optional<U>&& rhs);
```

21 **Effects:** See Table 43. The result of the expression bool(rhs) remains unchanged.

<table>
<thead>
<tr>
<th></th>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>assigns std::move(*rhs) to the contained value</td>
<td>initializes the contained value as if direct-non-list-initializing an object of type T with std::move(*rhs)</td>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>destroys the contained value by calling val-&gt;T::~T()</td>
<td>no effect</td>
</tr>
</tbody>
</table>

22 **Returns:** *this.

23 **Ensures:** bool(rhs) == bool(*this).

24 **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. This function shall not participate in overload resolution unless

(24.1) \(\text{is\_constructible\_v<T, U> is true,}\)
(24.2) \(\text{is\_assignable\_v<T&, U> is true,}\)
(24.3) \(\text{is\_constructible\_v<T, optional<U>&> is false,}\)
(24.4) \(\text{is\_constructible\_v<T, optional<U>&&> is false,}\)
(24.5) \(\text{is\_constructible\_v<T, const optional<U>&> is false,}\)
(24.6) \(\text{is\_constructible\_v<T, const optional<U>&&> is false,}\)
(24.7) \(\text{is\_convertible\_v<optional<U>&, T> is false,}\)
(24.8) \(\text{is\_convertible\_v<optional<U>&&}, T> is false,}\)
(24.9) \(\text{is\_convertible\_v<const optional<U>&, T> is false,}\)
(24.10) \(\text{is\_convertible\_v<const optional<U>&&}, T> is false,}\)
(24.11) \(\text{is_assignable_v<T&}, \text{optional<U>>&} \) is false,
(24.12) \(\text{is_assignable_v<T&}, \text{optional<U>>&} \) is false,
(24.13) \(\text{is_assignable_v<T&}, \text{const optional<U>>&} \) is false, and
(24.14) \(\text{is_assignable_v<T&}, \text{const optional<U>>&} \) is false.

\[
\text{template<class... Args> T& emplace(Args&&... args);
}
\]

\[
\begin{align*}
\text{Requires: } &\text{is_constructible_v<T, Args&&...> is true.} \\
\text{Effects: } &\text{Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing} \\
&\text{an object of type T with the arguments std::forward<Args>(args)....} \\
\text{Ensures: } &\text{*this contains a value.} \\
\text{Returns: } &\text{A reference to the new contained value.} \\
\text{Throws: } &\text{Any exception thrown by the selected constructor of T.} \\
\text{Remarks: } &\text{If an exception is thrown during the call to T's constructor, *this does not contain a value,} \\
&\text{and the previous *val (if any) has been destroyed.}
\end{align*}
\]

\[
\text{template<class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);
}
\]

\[
\begin{align*}
\text{Effects: } &\text{Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing} \\
&\text{an object of type T with the arguments il, std::forward<Args>(args)....} \\
\text{Ensures: } &\text{*this contains a value.} \\
\text{Returns: } &\text{A reference to the new contained value.} \\
\text{Throws: } &\text{Any exception thrown by the selected constructor of T.} \\
\text{Remarks: } &\text{If an exception is thrown during the call to T's constructor, *this does not contain a value,} \\
&\text{and the previous *val (if any) has been destroyed. This function shall not participate in overload} \\
&\text{resolution unless is_constructible_v<T, initializer_list<U>&, Args&&...> is true.}
\end{align*}
\]

20.6.3.4 Swap \[\text{optional.swap}\]

\[
\text{void swap(optional& rhs) noexcept(see below);} \\
\text{Requires: } \text{Lvalues of type T shall be swappable and is_move_constructible_v<T> is true.} \\
\text{Effects: } \text{See Table 44.}
\]

\[
\text{Table 44 — optional::swap(optional&): effects}
\]

<table>
<thead>
<tr>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>calls swap(*(*this), *rhs)</td>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>initializes the contained value of rhs as if direct-non-list-initializing an object of type T with the expression std::move(*(*this)), followed by val-&gt;T::~T(); postcondition is that *this does not contain a value and rhs contains a value</td>
</tr>
</tbody>
</table>
3   **Throws:** Any exceptions thrown by the operations in the relevant part of Table 44.
4   **Remarks:** The expression inside `noexcept` is equivalent to:
5   
6   ```cpp
7   is_nothrow_move_constructible_v<T> && is_nothrow_swappable_v<T>
8   ```
9   
10  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.

### 20.6.3.5 Observers

`constexpr const T* operator->() const;`  
`constexpr T* operator->();`

1   **Requires:** `*this` contains a value.
2   **Returns:** `val`.
3   **Throws:** Nothing.
4   **Remarks:** These functions shall be constexpr functions.

`constexpr const T& operator*() const&;`  
`constexpr T& operator*() &;`

5   **Requires:** `*this` contains a value.
6   **Returns:** `*val`.
7   **Throws:** Nothing.
8   **Remarks:** These functions shall be constexpr functions.

`constexpr T&& operator*() &&;`  
`constexpr const T&& operator*() const&&;`

9   **Requires:** `*this` contains a value.
10  **Effects:** Equivalent to:
11    ```cpp
12    return std::move(*val);
13    ```
14  
15  `constexpr explicit operator bool() const noexcept;`

16  **Returns:** `true` if and only if `*this` contains a value.
17  **Remarks:** This function shall be a constexpr function.

`constexpr bool has_value() const noexcept;`

18  **Returns:** `true` if and only if `*this` contains a value.
19  **Remarks:** This function shall be a constexpr function.

`constexpr const T& value() const&;`  
`constexpr T& value() &;`

20  **Effects:** Equivalent to:
21    ```cpp
22    return bool(*this) ? *val : throw bad_optional_access();
23    ```
24  
25  `constexpr T&& value() &&;`  
`constexpr const T&& value() const&&;`

26  **Effects:** Equivalent to:
27    ```cpp
28    return bool(*this) ? std::move(*val) : throw bad_optional_access();
29    ```
30  
31  `template<class U> constexpr T value_or(U&& v) const&;`

32  **Effects:** Equivalent to:
33    ```cpp
34    return bool(*this) ? **this : static_cast<T>(_U&& v) : throw bad_optional_access();
35    ```
36  
37  **Remarks:** If `is_copy_constructible_v<T> && is_convertible_v<U&&, T>`, the program is ill-formed.
template<class U> constexpr T value_or(U&& v) &&;

Effects: Equivalent to:
return bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v));

Remarks: If is_move_constructible_v<T> && is_convertible_v<U&&, T> is false, the program is ill-formed.

20.6.3.6 Modifiers

void reset() noexcept;

Effects: If *this contains a value, calls val->T::~T() to destroy the contained value; otherwise no effect.

Ensures: *this does not contain a value.

20.6.4 No-value state indicator

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

class bad_optional_access : public exception {
public:
    bad_optional_access();
};

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.

bad_optional_access();

Effects: Constructs an object of class bad_optional_access.

Ensures: what() returns an implementation-defined NTBS.

20.6.6 Relational operators

template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);

Requires: The expression *x == *y shall be well-formed and its result shall be 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 shall be constexpr functions.

template<class T, class U> constexpr bool operator!=(const optional<T>& x, const optional<U>& y);

Requires: The expression *x != *y shall be well-formed and its result shall be 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 shall be constexpr functions.

template<class T, class U> constexpr bool operator<(const optional<T>& x, const optional<U>& y);

Requires: *x < *y shall be well-formed and its result shall be 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 shall be constexpr functions.
template<class T, class U> constexpr bool operator>(const optional<T>& x, const optional<U>& y);

Requires: The expression \( \ast x > \ast y \) shall be well-formed and its result shall be convertible to bool.

Returns: If \( \ast x \), false; otherwise, if \( \ast y \), true; otherwise \( \ast x > \ast y \).

Remarks: Specializations of this function template for which \( \ast x > \ast y \) is a core constant expression shall be constexpr functions.

template<class T, class U> constexpr bool operator<=(const optional<T>& x, const optional<U>& y);

Requires: The expression \( \ast x \leq \ast y \) shall be well-formed and its result shall be convertible to bool.

Returns: If \( \ast x \), true; otherwise, if \( \ast y \), false; otherwise \( \ast x \leq \ast y \).

Remarks: Specializations of this function template for which \( \ast x \leq \ast y \) is a core constant expression shall be constexpr functions.

template<class T, class U> constexpr bool operator>=(const optional<T>& x, const optional<U>& y);

Requires: The expression \( \ast x \geq \ast y \) shall be well-formed and its result shall be convertible to bool.

Returns: If \( \ast y \), true; otherwise, if \( \ast x \), false; otherwise \( \ast x \geq \ast y \).

Remarks: Specializations of this function template for which \( \ast x \geq \ast y \) is a core constant expression shall be constexpr functions.

20.6.7 Comparison with nullopt

[optional.nullops]

template<class T> constexpr bool operator==(const optional<T>& x, nullopt_t) noexcept;
template<class T> constexpr bool operator==(nullopt_t, const optional<T>& x) noexcept;

Returns: \(!x\).

template<class T> constexpr bool operator!=(const optional<T>& x, nullopt_t) noexcept;
template<class T> constexpr bool operator!=(nullopt_t, const optional<T>& x) noexcept;

Returns: \(bool(x)\).

template<class T> constexpr bool operator<(const optional<T>& x, nullopt_t) noexcept;
template<class T> constexpr bool operator<(nullopt_t, const optional<T>& x) noexcept;

Returns: \(false\).

template<class T> constexpr bool operator>=(const optional<T>& x, nullopt_t) noexcept;
template<class T> constexpr bool operator>=(nullopt_t, const optional<T>& x) noexcept;

Returns: \(false\).

20.6.8 Comparison with T

[optional.comp.with.t]

template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);

Requires: The expression \( \ast x == \ast y \) shall be well-formed and its result shall be convertible to bool.

[Note: \( T \) need not be Cpp17EqualityComparable. — end note]
Effects: Equivalent to: \( \text{return bool}(x) \ ? \ *x == v : false; \)

template\langle class T, class U\rangle constexpr bool operator==(const T& v, const optional<U>& x);

Requires: The expression \( v == *x \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ v == *x : false; \)

template\langle class T, class U\rangle constexpr bool operator!=(const optional<T>& x, const U& v);

Requires: The expression \( *x != v \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ *x != v : true; \)

template\langle class T, class U\rangle constexpr bool operator!=(const T& v, const optional<U>& x);

Requires: The expression \( v != *x \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ v != *x : true; \)

template\langle class T, class U\rangle constexpr bool operator<(const optional<T>& x, const U& v);

Requires: The expression \( *x < v \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ *x < v : true; \)

template\langle class T, class U\rangle constexpr bool operator<(const T& v, const optional<U>& x);

Requires: The expression \( v < *x \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ v < *x : false; \)

template\langle class T, class U\rangle constexpr bool operator<=(const optional<T>& x, const U& v);

Requires: The expression \( *x <= v \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ *x <= v : true; \)

template\langle class T, class U\rangle constexpr bool operator<=(const T& v, const optional<U>& x);

Requires: The expression \( v <= *x \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ v <= *x : false; \)

template\langle class T, class U\rangle constexpr bool operator>=(const optional<T>& x, const U& v);

Requires: The expression \( *x >= v \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ *x >= v : false; \)

template\langle class T, class U\rangle constexpr bool operator>=(const T& v, const optional<U>& x);

Requires: The expression \( v >= *x \) shall be well-formed and its result shall be convertible to \( \text{bool} \).

Effects: Equivalent to: \( \text{return bool}(x) \ ? \ v >= *x : true; \)

\section{Specialized algorithms [optional.specalg]}

\subsection{20.6.9 void swap(optional\langle T\rangle& x, optional\langle T\rangle& y) noexcept(noexcept(x.swap(y)))

Effects: Calls x.swap(y).

Remarks: This function shall not participate in overload resolution unless \text{is\_move\_constructible\_}\langle\langle T\rangle\rangle is \text{true} and \text{is\_swappable\_}\langle\langle T\rangle\rangle is \text{true}.

Returns: optional\langle\text{optional}\langle T\rangle\rangle(\text{std\::forward}\langle\langle T\rangle\rangle(v)).
template<class T, class... Args>
constexpr optional<T> make_optional(Args&&... args);

Effects: Equivalent to: return optional<T>(in_place, std::forward<Args>(args)...);

template<class T, class U, class... Args>
constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);

Effects: Equivalent to: return optional<T>(in_place, il, std::forward<Args>(args)...);

20.6.10 Hash support

template<class T> struct hash<optional<T>>;

The specialization hash<optional<T>> is enabled (20.14.18) 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) shall evaluate to the same value as hash<remove_const_t<T>>(*o); otherwise it evaluates to an unspecified value. The member functions are not guaranteed to be noexcept.

20.7 Variants

20.7.1 In general

A variant object holds and manages the lifetime of a value. If the variant holds a value, that value’s type has to be one of the template argument types given to variant. These template arguments are called alternatives.

20.7.2 Header <variant> synopsis

namespace std {

// 20.7.3, class template
template<class... Types>
class variant;

// 20.7.4, variant helper classes
template<class T> struct variant_size;  
 // not defined
template<class T> struct variant_size<const T>;
template<class T> struct variant_size<volatile T>;
template<class T> struct variant_size<volatile const 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 T>
struct variant_alternative;  
 // not defined

template<
size_t I, class T> 
struct variant_alternative<I, const T>;

template<
size_t I, class T> 
struct variant_alternative<I, volatile T>;

template<
size_t I, class T> 
struct variant_alternative<I, volatile const T>;

template<
size_t I, class T> 
using variant_alternative_t = typename variant_alternative<I, T>::type;

template<
size_t I, class... Types>
struct variant_alternative<I, variant<Types...>>;

inline constexpr size_t variant_npos = -1;

// 20.7.5, value access
template<class T, class... Types>
constexpr bool holds_alternative(const variant<Types...>&) noexcept;

template<
size_t I, class... Types>
constexpr bool holds_alternative(const variant<Types...>& get(variant<Types...>&);  

template<
size_t I, class... Types>
constexpr variant_alternative_t& get(variant<Types...> &);  

template<
size_t I, class... Types>
constexpr variant_alternative_t& get(variant<Types...>&&);  

template<
size_t I, class... Types>
constexpr variant_alternative_t& get(variant<Types...>&&);  

template<
size_t I, class... Types>
constexpr variant_alternative_t& get(const variant<Types...>&);  

§ 20.7.2
template<
  size_t I, class... Types>
constexpr const variant_alternative_t<I, variant<Types...>>&&
get(const 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;

// 20.7.6, relational operators

template<class... Types>
constexpr bool operator==(const variant<Types...>&, const variant<Types...>&);

template<class... Types>
constexpr bool operator!=(const variant<Types...>&, const variant<Types...>&);

template<class... Types>
constexpr bool operator<(const variant<Types...>&, const variant<Types...>&);

template<class... Types>
constexpr bool operator>(const variant<Types...>&, const variant<Types...>&);

template<class... Types>
constexpr bool operator<=(const variant<Types...>&, const variant<Types...>&);

template<class... Types>
constexpr bool operator>=(const variant<Types...>&, const variant<Types...>&);

// 20.7.7, visitation

template<class Visitor, class... Variants>
constexpr see below
visit(Visitor&&, Variants&&...);

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 bool operator!=(monostate, monostate) noexcept;
constexpr bool operator<(monostate, monostate) noexcept;
constexpr bool operator>(monostate, monostate) noexcept;
constexpr bool operator<=(monostate, monostate) noexcept;
constexpr bool 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;
20.7.3 Class template variant

```cpp
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.
In the descriptions that follow, let $i$ be in the range $[0, \text{sizeof}\ldots\text{Types})$, and $T_i$ be the $i$th type in Types.

```cpp
constexpr variant() noexcept(see below);
```

**Effects:** Constructs a variant holding a value-initialized value of type $T_0$.

**Ensures:** `valueless_by_exception()` is false and `index()` is 0.

**Throws:** Any exception thrown by the value-initialization of $T_0$.

**Remarks:** This function shall be `constexpr` if and only if the value-initialization of the alternative type $T_0$ would satisfy the requirements for a `constexpr` function. The expression inside `noexcept` is equivalent to `is_nothrow_default_constructible_v<T_0>`. This function shall not participate in overload resolution unless `is_default_constructible_v<T_0>` is true. [Note: See also class `monostate`. — end note]

```cpp
constexpr variant(const variant& w);
```

**Effects:** If $w$ holds a value, initializes the variant to hold the same alternative as $w$ and direct-initializes the contained value with `get<j>(w)`, where $j$ is $w$.index(). Otherwise, initializes the variant to not hold a value.

**Throws:** Any exception thrown by direct-initializing any $T_i$ for all $i$.

**Remarks:** This constructor shall be defined as deleted unless `is_copy_constructible_v<T_i>` is true for all $i$. If `is_trivially_copy_constructible_v<T_i>` is true for all $i$, this constructor is trivial.

```cpp
constexpr variant(variant&& w) noexcept(see below);
```

**Effects:** If $w$ holds a value, initializes the variant to hold the same alternative as $w$ and direct-initializes the contained value with `get<j>(std::move(w))`, where $j$ is $w$.index(). 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 `is_nothrow_move_constructible_v<T_i>` for all $i$. This function shall not participate in overload resolution unless `is_move_constructible_v<T_i>` is true for all $i$. If `is_trivially_move_constructible_v<T_i>` is true for all $i$, this constructor is trivial.

```cpp
template<class T> constexpr variant(T&& t) noexcept;
```

Let $T_j$ be a type that is determined as follows: build an imaginary function `FUN(T_j)` for each alternative type $T_j$ for which $T_j = \{\text{std::forward<T>(t)}\}$; is well-formed for some invented variable $x$ and, if $T_j$ is `ce bool remove_cvref_t<T>` is bool. The overload `FUN(T_j)` selected by overload resolution for the expression `FUN(std::forward<T>(t))` defines the alternative $T_j$ which is the type of the contained value after construction.

**Effects:** Initializes `*this` to hold the alternative type $T_j$ and direct-initializes the contained value as if direct-non-list-initializing it with `std::forward<T>(t)`.

**Ensures:** `holds_alternative<T_j>(*this)` is true.

**Throws:** Any exception thrown by the initialization of the selected alternative $T_j$.

**Remarks:** This function shall not participate in overload resolution unless

1. `sizeof\ldots\text{Types})` is nonzero,
2. `is_same_v<remove_cvref_t<T>, \text{variant}>` is false,
3. `remove_cvref_t<T>` is neither a specialization of `in_place_type_t` nor a specialization of `in_place_index_t`,
4. `is_constructible_v<T_j, T>` is true, and
5. the expression `FUN(std::forward<T>(t))` (with `FUN` being the above-mentioned set of imaginary functions) is well-formed.
Note:

```cpp
variant<string, string> v("abc");
```

is ill-formed, as both alternative types have an equally viable constructor for the argument. — end note

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 shall be a constexpr constructor.

```cpp
template<class T, class... Arges> constexpr explicit variant(in_place_type_t<T>, Arges&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Arge>(args)....

Ensures: holds_alternative<T>(*this) is true.

Throws: Any exception thrown by calling the selected constructor of T.

Remarks: This function shall not participate in overload resolution unless there is exactly one occurrence of T in Types and is_constructible_v<T, Arge...> is true. If T’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```cpp
template<class T, class U, class... Arges>
constexpr explicit variant(in_place_type_t<T>, initializer_list<U> il, Arges&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Arge>(args)....

Ensures: holds_alternative<T>(*this) is true.

Throws: Any exception thrown by calling the selected constructor of T.

Remarks: This function shall not participate in overload resolution unless there is exactly one occurrence of T in Types and is_constructible_v<T, initializer_list<U>&, Arge...> is true. If T’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```cpp
template<size_t I, class... Arges>
constexpr explicit variant(in_place_index_t<I>, Arges&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T_I with the arguments std::forward<Arge>(args)....

Ensures: index() is I.

Throws: Any exception thrown by calling the selected constructor of T_I.

Remarks: This function shall not participate in overload resolution unless

(30.1) I is less than sizeof...(Types) and

(30.2) is_constructible_v<T_I, Arge...> is true.

If T_I’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```cpp
template<size_t I, class U, class... Arges>
constexpr explicit variant(in_place_index_t<I>, initializer_list<U> il, Arges&&... args);
```

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T_I with the arguments il, std::forward<Arge>(args)....

Ensures: index() is I.

Remarks: This function shall not participate in overload resolution unless

(33.1) I is less than sizeof...(Types) and

(33.2) is_constructible_v<T_I, initializer_list<U>&, Arge...> is true.

If T_I’s selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

20.7.3.2 Destructor

```cpp
~variant();
```

Effects: If valueless_by_exception() is false, destroys the currently contained value.

Remarks: If is_trivially_destructible_v<T_I> is true for all T_I, then this destructor is trivial.
### Assignment

```cpp
constexpr variant& operator=(const variant& rhs);
```

1. Let \( j \) be \( \text{rhs.index()} \).

2. **Effects:**

   - (2.1) If neither \( \ast \text{this} \) nor \( \text{rhs} \) holds a value, there is no effect.
   - (2.2) Otherwise, if \( \ast \text{this} \) holds a value but \( \text{rhs} \) does not, destroys the value contained in \( \ast \text{this} \) and sets \( \ast \text{this} \) to not hold a value.
   - (2.3) Otherwise, if \( \text{index()} == j \), assigns the value contained in \( \text{rhs} \) to the value contained in \( \ast \text{this} \).
   - (2.4) Otherwise, if either \( \text{is_nothrow_copy_constructible_v<T}\{j\}> \) is true or \( \text{is_nothrow_move_constructible_v<T}\{j\}> \) is false, equivalent to \( \text{emplace}\langle j\rangle(\text{get}\langle j\rangle(\text{rhs})) \).
   - (2.5) Otherwise, equivalent to \( \text{operator=} (\text{variant}(\text{rhs})) \).

3. **Returns:** \( \ast \text{this} \).

4. **Ensures:** \( \text{index()} == \text{rhs.index()} \).

5. **Remarks:** This operator shall be defined as deleted unless \( \text{is_copy_constructible_v<T}\{i\}> \) \&\& \( \text{is_copy_assignable_v<T}\{i\}> \) is true for all \( i \). If \( \text{is_trivially_copy_constructible_v<T}\{i\}> \) \&\& \( \text{is_trivially_copy_assignable_v<T}\{i\}> \) \&\& \( \text{is_trivially_destructible_v<T}\{i\}> \) is true for all \( i \), this assignment operator is trivial.

```cpp
constexpr variant& operator=(variant&& rhs) noexcept(
```

6. Let \( j \) be \( \text{rhs.index()} \).

7. **Effects:**

   - (7.1) If neither \( \ast \text{this} \) nor \( \text{rhs} \) holds a value, there is no effect.
   - (7.2) Otherwise, if \( \ast \text{this} \) holds a value but \( \text{rhs} \) does not, destroys the value contained in \( \ast \text{this} \) and sets \( \ast \text{this} \) to not hold a value.
   - (7.3) Otherwise, if \( \text{index()} == j \), assigns \( \text{get}\langle j\rangle(\text{std::move}(\text{rhs})) \) to the value contained in \( \ast \text{this} \).
   - (7.4) Otherwise, equivalent to \( \text{emplace}\langle j\rangle(\text{get}\langle j\rangle(\text{std::move}(\text{rhs}))) \).

8. **Returns:** \( \ast \text{this} \).

9. **Remarks:** This function shall not participate in overload resolution unless \( \text{is_move_constructible_v<T}\{i\}> \) \&\& \( \text{is_move_assignable_v<T}\{i\}> \) is true for all \( i \). If \( \text{is_trivially_move_constructible_v<T}\{i\}> \) \&\& \( \text{is_trivially_move_assignable_v<T}\{i\}> \) \&\& \( \text{is_trivially_destructible_v<T}\{i\}> \) is true for all \( i \), this assignment operator is trivial. The expression inside noexcept is equivalent to: \( \text{is_nothrow_move_constructible_v<T}\{i\}> \) \&\& \( \text{is_nothrow_move_assignable_v<T}\{i\}> \) \&\& \( \text{is_nothrow_move_constructible_v<T}\{i\}> \) for all \( i \).

10. **Effects:**

    - (11.1) If an exception is thrown during the call to \( T_j \)'s move construction (with \( j \) being \( \text{rhs.index()} \)), the \( \text{variant} \) will hold no value.
    - (11.2) 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; \( \text{index()} \) will be \( j \).

```cpp
template<class T> variant& operator=(T&& t) noexcept(
```

11. **Effects:**

    - (11.1) If \( \ast \text{this} \) holds a \( T_j \), assigns \( \text{std::forward<T}(t) \) to the value contained in \( \ast \text{this} \).
    - (11.2) Otherwise, if \( \text{is_nothrow_constructible_v<T}_j, T> \) \&\& \( \text{is_nothrow_move_constructible_v<T}_j, T> \) is true, equivalent to \( \text{emplace}\langle j\rangle(\text{std::forward<T}(t)) \).
    - (11.3) Otherwise, equivalent to \( \text{operator=} (\text{variant}(\text{std::forward<T}(t))) \).
Ensures: \texttt{holds_alternative}<T_j>(*\texttt{this}) is \texttt{true}, with \(T_j\) selected by the imaginary function overload resolution described above.

Remarks: \texttt{this}.

— \texttt{is_same_v<remove_cvref_t<T>, variant> is false},

— \texttt{is_assignable_v<T, T_\& T \& is_constructible_v<T_j, T> is true, and}

— the expression \texttt{FUN(std::forward<T>(t))} (with \texttt{FUN} being the above-mentioned set of imaginary functions) is well-formed.

\begin{itemize}
\item \texttt{variant<string, string> v;

\vspace{-20pt}
\end{itemize}

is ill-formed, as both alternative types have an equally viable constructor for the argument. \textit{— end note}

The expression inside \texttt{noexcept} is equivalent to:

\begin{itemize}
\item \texttt{is_nothrow_assignable_v<T_\& T \& is_nothrow_constructible_v<T_j, T>}
\item \texttt{If an exception is thrown during the assignment of \texttt{std::forward<T>(t)} to the value contained in \texttt{this}, the state of the contained value and \(t\) are as defined by the exception safety guarantee of the assignment expression; \texttt{valueless_by_exception()} will be \texttt{false}.}
\item \texttt{If an exception is thrown during the initialization of the contained value, the \texttt{variant} object might not hold a value.}
\end{itemize}

\subsection*{20.7.3.4 Modifiers} \texttt{[variant.mod]}

\begin{itemize}
\item \texttt{template<class T, class... Args> T& \texttt{emplace(Args\&... args);} }
\item Let \(I\) be the zero-based index of \(T\) in \texttt{Types}.
\item \texttt{Effects: Equivalent to: return \texttt{emplace}(I)(std::forward<Args>(args)...);}
\item \texttt{Remarks: This function shall not participate in overload resolution unless \texttt{is_constructible_v<T, Args...> is true, and T occurs exactly once in Types.}}
\end{itemize}

\begin{itemize}
\item \texttt{template<class T, class U, class... Args> T& \texttt{emplace(initializer_list<U> il, Args\&... args);} }
\item Let \(I\) be the zero-based index of \(T\) in \texttt{Types}.
\item \texttt{Effects: Equivalent to: return \texttt{emplace}(I)(il, std::forward<Args>(args)...);}
\item \texttt{Remarks: This function shall not participate in overload resolution unless \texttt{is_constructible_v<T, initializer_list<U>\&, Args...> is true, and T occurs exactly once in Types.}}
\end{itemize}

\begin{itemize}
\item \texttt{template<size_t I, class... Args> variant_alternative_t<I, variant< Types...>>\& \texttt{emplace(Args\&... args);} }
\item \texttt{Requires: I < sizeof...(Types).}
\item \texttt{Effects: Destroys the currently contained value if \texttt{valueless_by_exception()} is \texttt{false}. Then initializes the contained value as if direct-non-list-initializing a value of type \(T_j\) with the arguments \texttt{std::forward<Args>(args)....}}
\item \texttt{Ensures: \texttt{index()} is I.}
\item \texttt{Returns: A reference to the new contained value.}
\item \texttt{Threws: Any exception thrown during the initialization of the contained value.}
\item \texttt{Remarks: This function shall not participate in overload resolution unless \texttt{is_constructible_v<T_j, Args...> is true. If an exception is thrown during the initialization of the contained value, the \texttt{variant} might not hold a value.}}
\end{itemize}

\begin{itemize}
\item \texttt{template<size_t I, class U, class... Args> variant_alternative_t<I, variant< Types...>>\& \texttt{emplace(initializer_list<U> il, Args\&... args);} }
\item \texttt{Requires: I < sizeof...(Types).}
\end{itemize}
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; with the arguments il, std::forward<Args>(args)....

Ensures: index() is I.

Returns: A reference to the new contained value.

Throws: Any exception thrown during the initialization of the contained value.

Remarks: This function shall not participate in overload resolution unless is_constructible_v<T_i, initializer_list<U>&, Args...> is true. 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

struct S { operator int() { throw 42; };
variant<float, int> v(12.f);
v.emplace<0>(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);

Requires: Lvalues of type T_i shall be swappable (16.5.3.2) and is_move_constructible_v<T_i> shall be true for all i.

Effects:

(2.1) — If valueless_by_exception() && rhs.valueless_by_exception() no effect.

(2.2) — Otherwise, if index() == rhs.index(), calls swap(get<i>(*this), get<i>(rhs)) where i is index().

(2.3) — 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;

Remarks: All specializations of variant_size shall satisfy 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>;
template<class T> class variant_size<volatile T>;
template<class T> class variant_size<const volatile T>;

Let VS denote \( \text{variant\_size}<T> \) of the cv-unqualified type \( T \). Then each of the three templates shall satisfy the \( \text{Cpp17UnaryTypeTrait} \) requirements (20.15.1) with a base characteristic of \( \text{integral\_constant}<\text{size\_t}, \text{VS}::\text{value}> \).

```cpp
template<class... Types>
struct variant_size<variant<Types...>> : integral_constant<size_t, sizeof...(Types)> { }
```

```cpp
template<size_t I, class T> class variant_alternative<I, const T>;
```

```cpp
template<size_t I, class T> class variant_alternative<I, volatile T>;
```

```cpp
template<size_t I, class T> class variant_alternative<I, const volatile T>;
```

Let VA denote \( \text{variant\_alternative}<I, T> \) of the cv-unqualified type \( T \). Then each of the three templates shall meet the \( \text{Cpp17TransformationTrait} \) requirements (20.15.1) with a member typedef \( \text{type} \) that names the following type:

1. (3.1) for the first specialization, \( \text{add\_const\_t<VA::type>} \),
2. (3.2) for the second specialization, \( \text{add\_volatile\_t<VA::type>} \), and
3. (3.3) for the third specialization, \( \text{add\_cv\_t<VA::type>} \).

```cpp
variant_alternative<I, variant<Types...>>::type
```

**Requires:** \( I < \text{sizeof}\ldots(\text{Types}) \). The program is ill-formed if \( I \) is out of bounds.

**Value:** The type \( T \).

### 20.7.5 Value access

```cpp
template<class T, class... Types>
constexpr bool holds_alternative(const variant<Types...>& v) noexcept;
```

**Requires:** The type \( T \) occurs exactly once in \( \text{Types} \). Otherwise, the program is ill-formed.

**Returns:** \( \text{true} \) if \( \text{index()} \) is equal to the zero-based index of \( T \) in \( \text{Types} \).

```cpp
template<
size_t I, class
... Types>
constexpr variant_alternative_t<I, variant<Types...>>& get(variant<Types...>& v);
```

**Requires:** \( I < \text{sizeof}\ldots(\text{Types}) \). Otherwise, the program is ill-formed.

**Effects:** If \( v\_.\text{index()} \) is \( I \), returns a reference to the object stored in the \( \text{variant} \). Otherwise, throws an exception of type \( \text{bad\_variant\_access} \).

```cpp
template<class T, class... Types>
constexpr T& get(variant<Types...>& v);
```

**Requires:** The type \( T \) occurs exactly once in \( \text{Types} \). Otherwise, the program is ill-formed.

**Effects:** If \( v \) holds a value of type \( T \), returns a reference to that value. Otherwise, throws an exception of type \( \text{bad\_variant\_access} \).

```cpp
template<
size_t I, class
... Types>
constexpr add_pointer_t<variant_alternative_t<I, variant<Types...>>> get_if(variant<Types...>* v) noexcept;
```

**Requires:** \( I < \text{sizeof}\ldots(\text{Types}) \). Otherwise, the program is ill-formed.

**Returns:** A pointer to the value stored in the \( \text{variant} \), if \( v \_\!= \text{nullptr} \) and \( v\_->\text{index()} == I \). Otherwise, returns \( \text{nullptr} \).
template<class T, class... Types>
constexpr add_pointer_t<T> get_if(variant<Types...>* v) noexcept;

template<class T, class... Types>
constexpr add_pointer_t<const T> get_if(const variant<Types...>* v) noexcept;

Requires: The type T occurs exactly once in Types. Otherwise, the program is ill-formed.

Effects: Equivalent to: return get_if<i>(v); with i being the zero-based index of T in Types.

20.7.6 Relational operators

 template<class... Types>
constexpr bool operator==(const variant<Types...>& v, const variant<Types...>& w);

Requires: get<i>(v) == get<i>(w) is a valid expression returning a type 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);

Requires: get<i>(v) != get<i>(w) is a valid expression returning a type 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);

Requires: get<i>(v) < get<i>(w) is a valid expression returning a type 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);

Requires: get<i>(v) > get<i>(w) is a valid expression returning a type that is convertible to bool, for all i.

Returns: If v.valueless_by_exception(), false; otherwise if w.valueless_by_exception(), true; otherwise, if v.index() > w.index(), true; otherwise if v.index() < w.index(), false; otherwise get<i>(v) > get<i>(w) with i being v.index().

 template<class... Types>
constexpr bool operator<=(const variant<Types...>& v, const variant<Types...>& w);

Requires: get<i>(v) <= get<i>(w) is a valid expression returning a type 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);

Requires: get<i>(v) >= get<i>(w) is a valid expression returning a type 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().
20.7.7 Visitation

```cpp
template<class Visitor, class... Variants>
constexpr see below visit(Visitor&& vis, Variants&&... vars);
```

1 Let \( n \) be \( \text{sizeof}(...) \text{(Variants)} \). Let \( m \) be a pack of \( n \) values of type \text{size\_t}. Such a pack is called valid if \( 0 \leq m_i < \text{variant\_size\_v<remove\_reference\_t<Variants,>>} \) for all \( 0 \leq i < n \). For each valid pack \( m \), let \( e(m) \) denote the expression:

\[
\text{INVOLVE}(\text{std::forward<Visitor>(vis)}, \text{get}<m>(\text{std::forward<Variants>(vars)})\ldots) \quad \text{// see 20.14.3}
\]

for the first form and

\[
\text{INVOLVE}<R>(\text{std::forward<Visitor>(vis)}, \text{get}<m>(\text{std::forward<Variants>(vars)})\ldots) \quad \text{// see 20.14.3}
\]

for the second form.

2 Requires: For each valid pack \( m \), \( e(m) \) shall be a valid expression. All such expressions shall be of the same type and value category; otherwise, the program is ill-formed.

3 Returns: \( e(m) \), where \( m \) is the pack for which \( m_i \) is \text{vars\_i.index()} \) for all \( 0 \leq i < n \). The return type is \text{decltype}(e(m)) \) for the first form.

4 Throws: \text{bad\_variant\_access} if any variant in \text{vars} is \text{valueless\_by\_exception}().

5 Complexity: For \( n \leq 1 \), 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 \text{Variants}_0. For \( n > 1 \), the invocation of the callable object has no complexity requirements.

20.7.8 Class monostate

```cpp
struct monostate{};
```

1 The class \text{monostate} can serve as a first alternative type for a \text{variant} to make the \text{variant} type default constructible.

20.7.9 monostate relational operators

```cpp
constexpr bool operator==(monostate, monostate) noexcept { return true; }
constexpr bool operator!=(monostate, monostate) noexcept { return false; }
constexpr bool operator<(monostate, monostate) noexcept { return false; }
constexpr bool operator>(monostate, monostate) noexcept { return false; }
constexpr bool operator<=(monostate, monostate) noexcept { return true; }
constexpr bool operator>=(monostate, monostate) noexcept { return true; }
```

1 \[\text{Note: monostate objects have only a single state; they thus always compare equal. — end note}\]

20.7.10 Specialized algorithms

```cpp
template<class... Types>
void swap(variant<Types...>& v, variant<Types...>& w) noexcept(see below);
```

1 Effects: Equivalent to \text{v.swap(w)}.

2 Remarks: This function shall not participate in overload resolution unless \text{is\_move\_constructible\_v<T_i>} \&\& \text{is\_swappable\_v<T_i>} is true for all \( i \). The expression inside \text{noexcept} is equivalent to \text{noexcept(v.swap(w))}.

20.7.11 Class bad\_variant\_access

```cpp
class bad\_variant\_access : public exception {
public:
    bad\_variant\_access() noexcept;
    const char* what() const noexcept override;
};
```

1 Objects of type \text{bad\_variant\_access} are thrown to report invalid accesses to the value of a \text{variant} object.

2 Constructs a \text{bad\_variant\_access} object.
const char* what() const noexcept override;

Returns: An implementation-defined ntbs.

20.7.12 Hash support

template<class... Types> struct hash<variant<Types...>>;

1 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.

template<> struct hash<monostate>;

2 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>
    const T* any_cast(const any* operand) noexcept;

    template<class T>
    T* any_cast(any* operand) noexcept;
}

20.8.2 Class bad_any_cast

class bad_any_cast : public bad_cast {

    public:
    const char* what() const noexcept override;
};

1 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.
Remarks: The message may be a null-terminated multibyte string (16.4.2.2.5.2), suitable for conversion and display as a wstring (21.3, 28.4.1.4).

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;
    };
}

1 An object of class any stores an instance of any type that satisfies the constructor requirements or it has no value, and this is referred to as the state of the class any object. The stored instance is called the contained value. Two states are equivalent if either they both have no value, or they both have a value and the contained values are equivalent.

2 The non-member any_cast functions provide type-safe access to the contained value.

3 Implementations should avoid the use of dynamically allocated memory for a small contained value. However, any such small-object optimization shall only be applied to types T for which is_nothrow_move_constructible_v<T> is true. [Example: 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 Ensures: 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>, any_cast<const T&>(other)) where T is the type of the contained value.
3 \textit{Throws:} Any exceptions arising from calling the selected constructor for the contained value.

\begin{verbatim}
any(any&& other) noexcept;
\end{verbatim}

4 \textit{Effects:} If \textit{other}.\texttt{has_value()} is false, constructs an object that has no value. Otherwise, constructs an object of type \textit{any} that contains either the contained value of \textit{other}, or contains an object of the same type constructed from the contained value of \textit{other} considering that contained value as an rvalue.

5 \textit{Ensures:} \textit{other} is left in a valid but otherwise unspecified state.

\begin{verbatim}
template<class T>
any(T&& value);
\end{verbatim}

6 Let \(VT\) be \texttt{decay\_t<T>}.  

7 \textit{Requires:} \(VT\) shall satisfy the \texttt{Cpp17CopyConstructible} requirements. 

8 \textit{Effects:} Constructs an object of type \textit{any} that contains an object of type \(VT\) direct-initialized with \texttt{std::forward<T>(value)}. 

9 \textit{Remarks:} This constructor shall not participate in overload resolution unless \(VT\) is not the same type as \textit{any}, \(VT\) is not a specialization of \texttt{in\_place\_type\_t}, and \texttt{is\_copy\_constructible\_v<VT>} is true.

10 \textit{Throws:} Any exception thrown by the selected constructor of \(VT\).

\begin{verbatim}
template<class T, class... Args>
explicit any(in_place_type_t<T>, Args&&... args);
\end{verbatim}

11 Let \(VT\) be \texttt{decay\_t<T>}. 

12 \textit{Requires:} \(VT\) shall satisfy the \texttt{Cpp17CopyConstructible} requirements. 

13 \textit{Effects:} Initializes the contained value as if direct-non-list-initializing an object of type \(VT\) with the arguments \texttt{std::forward<Args>(args)}.... 

14 \textit{Ensures:} \(*\textit{this}\) contains a value of type \(VT\). 

15 \textit{Throws:} Any exception thrown by the selected constructor of \(VT\). 

16 \textit{Remarks:} This constructor shall not participate in overload resolution unless \texttt{is\_copy\_constructible\_v<VT>} is true and \texttt{is\_constructible\_v<VT, Args...> is true.}

\begin{verbatim}
template<class T, class U, class... Args>
explicit any(in_place_type_t<T>, initializer_list<U> il, Args&&... args);
\end{verbatim}

17 Let \(VT\) be \texttt{decay\_t<T>}. 

18 \textit{Requires:} \(VT\) shall satisfy the \texttt{Cpp17CopyConstructible} requirements. 

19 \textit{Effects:} Initializes the contained value as if direct-non-list-initializing an object of type \(VT\) with the arguments \(il, \texttt{std::forward<Args>(args)}\).... 

20 \textit{Ensures:} \(*\textit{this}\) contains a value. 

21 \textit{Throws:} Any exception thrown by the selected constructor of \(VT\). 

22 \textit{Remarks:} This constructor shall not participate in overload resolution unless \texttt{is\_copy\_constructible\_v<VT>} is true and \texttt{is\_constructible\_v<VT, initializer\_list<U>\&, Args...> is true.}

\begin{verbatim}
~any();
\end{verbatim}

23 \textit{Effects:} As if by \texttt{reset()}. 

\subsection{20.8.3.2 Assignment} \hfill \texttt{[any.assign]}

\begin{verbatim}
any\& operator=(const any\& rhs);
\end{verbatim}

1 \textit{Effects:} As if by \texttt{any(rhs).swap(*this)}. No effects if an exception is thrown. 

2 \textit{Returns:} \(*\textit{this}\). 

3 \textit{Throws:} Any exceptions arising from the copy constructor for the contained value.

\begin{verbatim}
any\& operator=(any&& rhs) noexcept;
\end{verbatim}

4 \textit{Effects:} As if by \texttt{any(std::move(rhs)).swap(*this)}. 

5 \textit{Returns:} \(*\textit{this}\). 

\section*{§ 20.8.3.2}
Ensures: The state of *this is equivalent to the original state of rhs and rhs is left in a valid but otherwise unspecified state.

```cpp
template<class T>
any& operator=(T&& rhs);
```

Let VT be decay_t<T>.

Requires: VT shall satisfy 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.

Remarks: This operator shall not participate in overload resolution unless VT is not the same type as any and is_copy_constructible_v<VT> is true.

Throws: Any exception thrown by the selected constructor of VT.

### 20.8.3.3 Modifiers

```cpp
template<class T, class... Args>
decay_t<T>& emplace(Args&&... args);
```

Let VT be decay_t<T>.

Requires: VT shall satisfy 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<Arge>(args)....

Ensures: *this contains a value.

Returns: A reference to the new contained value.

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. This function shall not participate in overload resolution unless is_copy_constructible_v<VT> is true and is_constructible_v<VT, Args...> is true.

```cpp
template<class T, class U, class... Args>
decay_t<T>& emplace(initializer_list<U> il, Args&&... args);
```

Let VT be decay_t<T>.

Requires: VT shall satisfy 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<Arge>(args)....

Ensures: *this contains a value.

Returns: A reference to the new contained value.

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. The function shall not participate in overload resolution unless is_copy_constructible_v<VT> is true and is_constructible_v<VT, initializer_list<U>&, Args...> is true.

```cpp
void reset() noexcept;
```

Effects: If has_value() is true, destroys the contained value.

Ensures: has_value() is false.

```cpp
void swap(any& rhs) noexcept;
```

Effects: Exchanges the states of *this and rhs.
20.8.3.4 Observers

```cpp
bool has_value() const noexcept;

1 Returns: true if *this contains an object, otherwise false.
```

```cpp
const type_info& type() const noexcept;
```

2 Returns: typeid(T) if *this has a contained value of type T, otherwise typeid(void).  

3 [Note: Useful for querying against types known either at compile time or only at runtime. — end note]

20.8.4 Non-member functions

```cpp
void swap(any& x, any& y) noexcept;
```

1 Effects: As if by x.swap(y).

```cpp
template<class T, class... Args>
any make_any(Args&& ...args);
```

2 Effects: Equivalent to: return any(in_place_type<T>, std::forward<Args>(args)...);

```cpp
template<class T, class U, class... Args>
any make_any(initializer_list<U> il, Args&& ...args);
```

3 Effects: Equivalent to: return any(in_place_type<T>, il, std::forward<Args>(args)...);

```cpp
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);
```

4 Let U be the type remove_cvref_t<T>.

5 Requires: 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. Otherwise the program is ill-formed.

6 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))).

7 Throws: bad_any_cast if operand.type() != typeid(remove_reference_t<T>.

8 [Example:]

```cpp
any x(5);
assert(any_cast<int>(x) == 5); // x holds int
any_cast<int&>(x) = 10; // cast to reference
assert(any_cast<int>(x) == 10);
```

```cpp
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);
```

```cpp
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");
```

```cpp
string cat("Meow");
const any y(cat);
assert(any_cast<const string&>(y) == cat);
```

```cpp
any_cast<string&>(y);
// error; cannot
// any_cast away const
```
template<class T>
const T* any_cast(const any* operand) noexcept;
template<class T>
T* any_cast(any* operand) noexcept;

Returns: If operand != nullptr && operand->type() == typeid(T), a pointer to the object contained by operand; otherwise, nullptr.

Example:
bool is_string(const any& operand) {
    return any_cast<string>(&operand) != nullptr;
}

— end example] 9

20.9 Bitsets

20.9.1 Header <bitset> synopsis

#include <string>
#include <iosfwd> // for istream (29.7.1), ostream (29.7.2), see 29.3.1

namespace std {
    template<size_t N> class bitset;
    
    // 20.9.4, bitset operators
    template<size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&) noexcept;
    template<size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&) noexcept;
    template<size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&) noexcept;
    template<class charT, class traits, size_t N>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
    template<class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}

The header <bitset> defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

20.9.2 Class template bitset

namespace std {
    template<size_t N> class bitset {
        public:
            // bit reference
            class reference {
                friend class bitset;
                reference() noexcept;
            
            public:
                reference(const reference&) = default;
                ~reference();
                reference operator=(bool x) noexcept; // for b[i] = x;
                reference operator=(const reference&) noexcept; // for b[i] = b[j];
                bool operator~() const noexcept; // flips the bit
                operator bool() const noexcept; // for x = b[i];
                reference flip() noexcept; // for b[i].flip();
            }
        
        // 20.9.2.1, constructors
        constexpr bitset() noexcept;
    }

§ 20.9.2
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 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:

1. an invalid-argument error is associated with exceptions of type `invalid_argument` (19.2.4);
2. an out-of-range error is associated with exceptions of type `out_of_range` (19.2.6);
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: Constructs an object of class `bitset<N>`, initializing all bits to zero.

constexpr bitset(unsigned long long val) noexcept;  // Effects: Constructs an object of class `bitset<N>`, initializing 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 (6.7) of `unsigned long long`. If M < N, the remaining bit positions are initialized to zero.

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'));

    // Throws: `out_of_range` if `pos > str.size()` or `invalid_argument` if an invalid character is found (see below).
    // Effects: Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size()` - `pos`. The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values. Otherwise, the function constructs an object of class `bitset<N>`, initializing 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.

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 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.

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.

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:
- If I < pos, the new value is zero;
- 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:
- If pos >= N - I, the new value is zero;
- 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);

Throws: out_of_range if pos does not correspond to a valid bit position.

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.

bitset<N>& reset() noexcept;

Effects: Resets all bits in *this.

Returns: *this.

bitset<N>& reset(size_t pos);

Throws: out_of_range if pos does not correspond to a valid bit position.

Effects: Resets the bit at position pos in *this.

Returns: *this.

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);

Throws: out_of_range if pos does not correspond to a valid bit position.
Effects: Toggles the bit at position \( pos \) in \(*this\).

Returns: \(*this\).

unsigned long to_ulong() const;

Throws: overflow_error if the integral value \( x \) corresponding to the bits in \(*this\) cannot be represented as type unsigned long.

Returns: \( x \).

unsigned long long to_ullong() const;

Throws: overflow_error if the integral value \( x \) corresponding to the bits in \(*this\) cannot be represented as type unsigned long long.

Returns: \( x \).

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 \( \text{zero} \), bit value one becomes the character \( \text{one} \).

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 operator!=(const bitset<N>& rhs) const noexcept;

Returns: true if \(!(*this == rhs)\).

bool test(size_t pos) const;

Throws: out_of_range if \( pos \) does not correspond to a valid bit position.

Returns: true if the bit at position \( pos \) in \(*this\) has the value one.

bool all() const noexcept;

Returns: \( \text{count}() == \text{size}() \).

bool any() const noexcept;

Returns: \( \text{count}() != 0 \).

bool none() const noexcept;

Returns: \( \text{count}() == 0 \).

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;

   Requires: pos shall be 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);

   Requires: pos shall be 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 (6.8.2), any access or update
   through the resulting reference potentially accesses or modifies, respectively, the entire underlying
   bitset.

20.9.3 bitset hash support

   [bitset.hash]

   template<size_t N> struct hash<bitset<N>>;

1   The specialization is enabled (20.14.18).

20.9.4 bitset operators

   [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);

4   A formatted input function (29.7.4.2).

5   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).

6   If no characters are stored in str, calls is.setstate(ios_base::failbit) (which may throw ios_base::failure
   (29.5.5.4)).

7   Returns: is.

template<class charT, class traits, size_t N>
   basic_ostream<charT, traits>&
      operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);

8   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

This subclause 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 multiple objects in uninitialized memory buffers (20.10.3–20.10.11). 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
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 Ptr>
    auto to_address(const Ptr& p) noexcept;
    template<class T>
    constexpr T* to_address(T* 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>
    auto uses_allocator_construction_args(const Alloc& alloc, Args&&... args) -> see below;
    template<class T, class Alloc, class Tuple1, class Tuple2>
    auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
                                          Tuple1&& x, Tuple2&& y) -> see below;
    template<class T, class Alloc>
    auto uses_allocator_construction_args(const Alloc& alloc) -> see below;
    template<class T, class Alloc, class U, class V>
    auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) -> see below;
    template<class T, class Alloc, class U, class V>
    auto uses_allocator_construction_args(const Alloc& alloc, const pair<U,V>& pr) -> see below;
    template<class T, class Alloc, class U, class V>
    auto uses_allocator_construction_args(const Alloc& alloc, pair<U,V>&& pr) -> see below;
```
template<class T, class Alloc, class... Args>
    T make_obj_using_allocator(const Alloc& alloc, Args&&... args);

template<class T, class Alloc, class... Args>
    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;

template<class T, class U>
    bool operator==(const allocator<T>&, const allocator<U>&) noexcept;

template<class T, class U>
    bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;

// 20.10.11, specialized algorithms

// 20.10.11.1, special memory concepts

namespace ranges {
    template<typename I, no-throw-input-iterator I, no-throw-sentinel<I> S>
        requires DefaultConstructible<iter_value_t<I>>
            I uninitialized_default_construct(I first, S last);

    template<typename I, no-throw-forward-range R>
        requires DefaultConstructible<iter_value_t<iterator_t<R>>>
            safe_iterator_t<R> uninitialized_default_construct(R&& r);

    template<typename I>
        requires DefaultConstructible<iter_value_t<I>>
            I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}

namespace ranges {
}

§ 20.10.2
template<class ExecutionPolicy, class ForwardIterator, class Size>
ForwardIterator uninitialized_value_construct_n(ExecutionPolicy&& exec, // see 25.3.5
   ForwardIterator first, Size n);

namespace ranges {

// see 25.3.5
}

namespace ranges {

// see 25.3.5
}

}
template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator> uninitialized_move_n(ExecutionPolicy&& exec, // see 25.3.5
        InputIterator first, Size n,
        ForwardIterator result);

namespace ranges {
    template<class I, class O>
        using uninitialized_move_result = uninitialized_copy_result<I, O>;
    template<InputIterator I, Sentinel<I> S1,
        no-throw-forward-iterator O,
        no-throw-sentinel<O> S2>
        requires Constructible<iter_value_t<O>, iter_rvalue_reference_t<I>>
        uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
    template<InputRange IR, no-throw-forward-range OR>
        requires Constructible<iter_value_t<iterator_t<OR>>,
            iter_rvalue_reference_t<iterator_t<IR>>>,
        uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_move(IR&& input_range, OR&& output_range);
    template<class I, class O>
        using uninitialized_move_n_result = uninitialized_copy_result<I, O>;
    template<InputIterator I,
        no-throw-forward-iterator O,
        no-throw-sentinel<O> S>
        requires Constructible<iter_value_t<O>, iter_rvalue_reference_t<I>>
        uninitialized_move_n_result<I, O>
        uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}

template<class ForwardIterator, class T>
        void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
    template<class ExecutionPolicy, class ForwardIterator, class T>
        void uninitialized_fill(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator first, ForwardIterator last, const T& x);
    template<class ForwardIterator, class Size, class T>
        ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
    template<class ExecutionPolicy, class ForwardIterator, class Size, class T>
        ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator first, Size n, const T& x);

namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
        requires Constructible<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<iter_value_t<iterator_t<R>>, const T&>
        safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);

    template<no-throw-forward-iterator I, class T>
        requires Constructible<iter_value_t<I>, const T&>
        I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}

template<class T>
        void destroy_at(T* location);
    template<class ForwardIterator>
        void destroy(ForwardIterator first, ForwardIterator last);
    template<class ExecutionPolicy, class ForwardIterator>
        void destroy(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator first, ForwardIterator last);
    template<class ForwardIterator, class Size>
        ForwardIterator destroy_n(ForwardIterator first, Size n);
    template<class ExecutionPolicy, class ForwardIterator, class Size>
        ForwardIterator destroy_n(ExecutionPolicy&& exec, // see 25.3.5
            ForwardIterator first, Size n);
namespace ranges {
    template<typename T>
    void destroy_at(T* location) noexcept;

    template<typename I, typename S>
    requires Destructible<iter_value_t<I>>
    I destroy(I first, S last) noexcept;

    template<typename R>
    requires Destructible<iter_value_t<iterator_t<R>>
    safe_iterator_t<R> destroy(R&& r) noexcept;

    template<typename I>
    requires Destructible<iter_value_t<I>>
    I destroy_n(I first, iter_difference_t<I> n) noexcept;
}

// 20.11.1, class template unique_ptr
template<typename T> struct default_delete;
template<typename T> struct default_delete<T[]>
    class unique_ptr {
        template<typename T, typename D = default_delete<T>>
        class unique_ptr {
            template<typename T, typename... Args>
            unique_ptr<T> make_unique(Args&&... args); // T is not array
            template<typename T, size_t n>
            unique_ptr<T> make_unique(size_t n); // T is U[]
            template<typename T, typename... Args>
            unspecified make_unique(Args&&...) = delete; // T is U[N]

            template<typename T>
            unique_ptr<T> make_unique_default_init(); // T is not array
            template<typename T, size_t n>
            unique_ptr<T> make_unique_default_init(size_t n); // T is U[]
            template<typename T, typename... Args>
            unspecified make_unique_default_init(Args&&...) = delete; // T is U[N]

            template<typename T, typename D>
            void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;
            template<typename T1, typename D1, typename T2, typename D2>
            bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
            template<typename T1, typename D1, typename T2, typename D2>
            bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
            template<typename T1, typename D1, typename T2, typename D2>
            bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
            template<typename T1, typename D1, typename T2, typename D2>
            bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
            template<typename T1, typename D1, typename T2, typename D2>
            bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
            template<typename T1, typename D1, typename T2, typename D2>
            bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
            template<typename T, typename D>
            bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
            template<typename T, typename D>
            bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;
            template<typename T, typename D>
            bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;
            template<typename T, typename D>
            bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;
            template<typename T, typename D>
            bool operator<(const unique_ptr<T, D>& x, nullptr_t);
            template<typename T, typename D>
            bool operator<(nullptr_t, const unique_ptr<T, D>& y);
            template<typename T, typename D>
            bool operator>(const unique_ptr<T, D>& x, nullptr_t);
            template<typename T, typename D>
            bool operator>(nullptr_t, const unique_ptr<T, D>& y);
            template<typename T, typename D>
            bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
            template<typename T, typename D>
            bool operator>=(nullptr_t, const unique_ptr<T, D>& y);
            template<typename T, typename D>
            bool operator<=(const unique_ptr<T, D>& x, nullptr_t);
            template<typename T, typename D>
            bool operator<=(nullptr_t, const unique_ptr<T, D>& y);
            template<typename T, typename D>
            bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
            template<typename T, typename D>
            bool operator>=(nullptr_t, const unique_ptr<T, D>& y);
        }
    }

§ 20.10.2
template<class T, class D>
    bool operator>(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
    bool operator>(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
    bool operator<(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
    bool operator<(nullptr_t, const unique_ptr<T, D>& y);
template<class T, class D>
    bool operator>=(const unique_ptr<T, D>& x, nullptr_t);
template<class T, class D>
    bool operator>=(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class 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[]

// 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>
    bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
bool 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>
bool operator==(nullptr_t, const shared_ptr<T>& y) noexcept;
template<class T>
bool operator!=(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
bool operator!=(nullptr_t, const shared_ptr<T>& y) noexcept;
template<class T>
bool operator<(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
bool operator<(nullptr_t, const shared_ptr<T>& y) noexcept;
template<class T>
bool operator>(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
bool operator>(nullptr_t, const shared_ptr<T>& y) noexcept;
template<class T>
bool operator<=(const shared_ptr<T>& x, nullptr_t) noexcept;
template<class T>
bool operator<=(nullptr_t, const shared_ptr<T>& y) 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>
shared_ptr<T> static_pointer_cast(shared_ptr<U>&& r) noexcept;
template<class T, class U>
shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
shared_ptr<T> dynamic_pointer_cast(shared_ptr<U>&& r) noexcept;
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;
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;

// 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>>;

// 20.11.8, atomic smart pointers
template<class T> struct atomic<shared_ptr<T>>;
template<class T> struct atomic<weak_ptr<T>>;

20.10.3 Pointer traits

The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

```cpp
namespace std {
    template<class Ptr> struct pointer_traits {
        using pointer = Ptr;
        using element_type = see below;
        using difference_type = see below;

        template<class U> using rebind = see below;

        static pointer pointer_to(see below r);
    };

    template<class T> struct pointer_traits<T*> {
        using pointer = T*;
        using element_type = T;
        using difference_type = ptrdiff_t;

        template<class U> using rebind = U*;

        static constexpr pointer pointer_to(see below r) noexcept;
    };
}
```

20.10.3.1 Member types

```cpp
using element_type = see below;

Type: `Ptr::element_type` if the qualified-id `Ptr::element_type` is valid and denotes a type (13.9.2); 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;

Type: `Ptr::difference_type` if the qualified-id `Ptr::difference_type` is valid and denotes a type (13.9.2); otherwise, `ptrdiff_t`.

template<class U> using rebind = see below;

Alias template: `Ptr::rebind<U>` if the qualified-id `Ptr::rebind<U>` is valid and denotes a type (13.9.2); 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 Remarks: If element_type is cv void, the type of r is unspecified; otherwise, it is element_type.
2 Returns: The first member function returns a pointer to r obtained by calling Ptr::pointer_to(r) through which indirection is valid; an instantiation of this function is ill-formed if Ptr does not have a matching pointer_to static member function. The second member function returns addressof(r).

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;

1 Returns: A pointer of type element_type* that references the same location as the argument p.
2 [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 Ptr> auto to_address(const Ptr& p) noexcept;

1 Returns: pointer_traits<Ptr>::to_address(p) if that expression is well-formed (see 20.10.3.3), otherwise to_address(p.operator->()).

template<class T> constexpr T* to_address(T* p) noexcept;

2 Requires: T is not a function type. Otherwise the program is ill-formed.
3 Returns: p.

20.10.5 Pointer safety

A complete object is declared reachable while the number of calls to declare_reachable with an argument referencing the object exceeds the number of calls to undeclare_reachable with an argument referencing the object.

void declare_reachable(void* p);

1 Requires: p shall be a safely-derived pointer (6.6.5.4.3) or a null pointer value.
2 Effects: If p is not null, the complete object referenced by p is subsequently declared reachable (6.6.5.4.3).
3 Throws: May throw bad_alloc if the system cannot allocate additional memory that may be required to track objects declared reachable.

template<class T> T* undeclare_reachable(T* p);

5 Requires: If p is not null, the complete object referenced by p shall have been previously declared reachable, and shall be live (6.6.3) 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 shall compare 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]

void declare_no_pointers(char* p, size_t n);

9 Requires: No bytes in the specified range are currently registered with declare_no_pointers(). If the specified range is in an allocated object, then it shall be entirely within a single allocated object. The object shall be 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 indirection through a pointer located there is undefined if the object it points to was 
created by global \texttt{operator new} and not previously declared reachable. \[ \text{Note: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — end note} \]

Throws: Nothing.

[\text{Note: Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — end note}]

\begin{verbatim}
void undeclare_no_pointers(char* p, size_t n);
\end{verbatim}

Requires: The same range shall previously have been passed to \texttt{declare_no_pointers()}.

Effects: Unregisters a range registered with \texttt{declare_no_pointers()} for destruction. It shall be called before the lifetime of the object ends.

Throws: Nothing.

\begin{verbatim}
pointer_safety get_pointer_safety() noexcept;
\end{verbatim}

Returns: \texttt{pointer_safety::strict} if the implementation has strict pointer safety (6.6.5.4.3). It is implementation-defined whether \texttt{get_pointer_safety} returns \texttt{pointer_safety::relaxed} or \texttt{pointer_safety::preferred} if the implementation has relaxed pointer safety.\footnote{\texttt{pointer_safety::preferred} might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.}

20.10.6 Pointer alignment [ptr.align]

\begin{verbatim}
void* align(size_t alignment, size_t size, void*& ptr, size_t& space);
\end{verbatim}

Effects: If it is possible to fit \texttt{size} bytes of storage aligned by \texttt{alignment} into the buffer pointed to by \texttt{ptr} with length \texttt{space}, the function updates \texttt{ptr} to represent the first possible address of such storage and decreases \texttt{space} by the number of bytes used for alignment. Otherwise, the function does nothing.

Requires:

(2.1) alignment shall be a power of two

(2.2) ptr shall represent the address of contiguous storage of at least \texttt{space} bytes

Returns: A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of \texttt{ptr}.

[\text{Note: The function updates its \texttt{ptr} and \texttt{space} arguments so that it can be called repeatedly with possibly different \texttt{alignment} and \texttt{size} arguments for the same buffer. — end note}]

\begin{verbatim}
template<size_t N, class T>
[[nodiscard]] constexpr T* assume_aligned(T* ptr);
\end{verbatim}

Mandates: \( N \) is a power of two.

Expects: \texttt{ptr} points to an object \( X \) of a type similar (7.3.5) to \( T \), where \( X \) has alignment \( N \) (6.6.6).

Returns: \texttt{ptr}.

Throws: Nothing.

[\text{Note: The alignment assumption on an object \( X \) expressed by a call to \texttt{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 \texttt{assume_aligned}. — end note}]

20.10.7 Allocator argument tag [allocator.tag]

namespace std {
struct allocator_arg_t { explicit allocator_arg_t() = default; };
inline constexpr allocator_arg_t allocator_arg{};
}

\footnote{\texttt{pointer_safety::preferred} might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.}
The `allocator_arg_t` struct is an empty class type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see tuple 20.5) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the `Cpp17Allocator` requirements (Table 34).

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 (13.9.2) 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>  
auto uses_allocator_construction_args(const Alloc& alloc, Args&&... args) -> 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, Alloc, Args...> is true, return tuple<allocator_arg_t, const Alloc&, Args&&...>( 
   allocator_arg, alloc, std::forward<Args>(args)...)
3. Otherwise, if `uses_allocator_v<T, Alloc> is true and is_constructible_v<T, Args..., Alloc> is true, return forward_as_tuple(std::forward<Args>(args)..., alloc).
4. 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]

template<class T, class Alloc, class Tuple1, class Tuple2>
auto uses_allocator_construction_args(const Alloc& alloc, piecewise_construct_t,
    Tuple1&& x, Tuple2&& y) -> see below;

**Constraints:** T is a specialization of pair.

**Effects:** For T specified as pair<T1, T2>, equivalent to:

\[
\begin{align*}
\text{return make_tuple(} & \\
& \text{piecewise_construct,} \\
& \text{apply([}&\text{alloc} (&\text{auto}\&\ldots \text{args1}) \{ & \\
& \text{return uses_allocator_construction_args<T1>(} & \\
& \text{alloc, std::forward<brule>dec-type(args1)>(args1)\ldots);} & \\
& \}, \text{std::forward<brule>Tuple1>(x)})}, & \\
& \text{apply([}&\text{alloc} (&\text{auto}\&\ldots \text{args2}) \{ & \\
& \text{return uses_allocator_construction_args<T2>(} & \\
& \text{alloc, std::forward<brule>dec-type(args2)>(args2)\ldots);} & \\
& \}, \text{std::forward<brule>Tuple2>(y)})});
\end{align*}
\]

template<class T, class Alloc>
auto uses_allocator_construction_args(const Alloc& alloc) -> see below;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

\[
\begin{align*}
\text{return uses_allocator_construction_args<T>(alloc, piecewise_construct,} & \\
& \text{tuple<>{}, tuple<>{}};
\end{align*}
\]

template<class T, class Alloc, class U, class V>
auto uses_allocator_construction_args(const Alloc& alloc, U&& u, V&& v) -> see below;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

\[
\begin{align*}
\text{return uses_allocator_construction_args<T>(alloc, piecewise_construct,} & \\
& \text{forward_as_tuple(std::forward<brule>U>(u)),} & \\
& \text{forward_as_tuple(std::forward<brule>V>(v))};
\end{align*}
\]

template<class T, class Alloc, class U, class V>
auto uses_allocator_construction_args(const Alloc& alloc, const pair<U, V>& pr) -> see below;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

\[
\begin{align*}
\text{return uses_allocator_construction_args<T>(alloc, piecewise_construct,} & \\
& \text{forward_as_tuple(pr.first),} & \\
& \text{forward_as_tuple(pr.second)});
\end{align*}
\]

template<class T, class Alloc, class U, class V>
auto uses_allocator_construction_args(const Alloc& alloc, pair<U, V>&& pr) -> see below;

**Constraints:** T is a specialization of pair.

**Effects:** Equivalent to:

\[
\begin{align*}
\text{return uses_allocator_construction_args<T>(alloc, piecewise_construct,} & \\
& \text{std::move(pr).first),} & \\
& \text{std::move(pr).second)});
\end{align*}
\]

template<class T, class Alloc, class... Args>
T make_obj_using_allocator(const Alloc& alloc, Args&&... args);

**Effects:** Equivalent to:

\[
\begin{align*}
\text{return make_from_tuple<T>(uses_allocator_construction_args<T>(} & \\
& \text{alloc, std::forward<brule>Args>(args)\ldots)});
\end{align*}
\]
template<class T, class Alloc, class... Args>
T* uninitialized_construct_using_allocator(T* p, const Alloc& alloc, Args&&... args);

Effects: Equivalent to:
return ::new(static_cast<void*>(p))
T(make_obj_using_allocator<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 pointer allocate(Alloc& a, size_type n);
        [[nodiscard]] static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
        static void deallocate(Alloc& a, pointer p, size_type n);
        template<class T, class... Args>
        static void construct(Alloc& a, T* p, Args&&... args);
        template<class T>
        static void destroy(Alloc& a, T* p);
        static size_type max_size(const Alloc& a) noexcept;
        static Alloc select_on_container_copy_construction(const Alloc& rhs);
    };
}

20.10.9.1 Member types

using pointer = see below;

Type: Alloc::pointer if the qualified-id Alloc::pointer is valid and denotes a type (13.9.2); otherwise, value_type*.

using const_pointer = see below;

Type: Alloc::const_pointer if the qualified-id Alloc::const_pointer is valid and denotes a type (13.9.2); otherwise, pointer_traits<pointer>::rebind<const value_type>.
using void_pointer = see below;  

Type: Alloc::void_pointer if the qualified-id Alloc::void_pointer is valid and denotes a type (13.9.2); otherwise, pointer_traits<pointer>::rebind<void>.

using const_void_pointer = see below;  

Type: Alloc::const_void_pointer if the qualified-id Alloc::const_void_pointer is valid and denotes a type (13.9.2); otherwise, pointer_traits<pointer>::rebind<const void>.

using difference_type = see below;  

Type: Alloc::difference_type if the qualified-id Alloc::difference_type is valid and denotes a type (13.9.2); otherwise, pointer_traits<pointer>::difference_type.

using size_type = see below;  

Type: Alloc::size_type if the qualified-id Alloc::size_type is valid and denotes a type (13.9.2); otherwise, make_unsigned_t<difference_type>.

using propagate_on_container_copy_assignment = see below;  

Type: Alloc::propagate_on_container_copy_assignment if the qualified-id Alloc::propagate_on_container_copy_assignment is valid and denotes a type (13.9.2); otherwise, false_type.

using propagate_on_container_move_assignment = see below;  

Type: Alloc::propagate_on_container_move_assignment if the qualified-id Alloc::propagate_on_container_move_assignment is valid and denotes a type (13.9.2); otherwise, false_type.

using propagate_on_container_swap = see below;  

Type: Alloc::propagate_on_container_swap if the qualified-id Alloc::propagate_on_container_swap is valid and denotes a type (13.9.2); otherwise, false_type.

using is_always_equal = see below;  

Type: Alloc::is_always_equal if the qualified-id Alloc::is_always_equal is valid and denotes a type (13.9.2); otherwise, is_empty<Alloc>::type.

template<class T> using rebind_alloc = see below;  

Alias template: Alloc::rebind<T>::other if the qualified-id Alloc::rebind<T>::other is valid and denotes a type (13.9.2); 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

[[nodiscard]] static pointer allocate(Alloc& a, size_type n);  

Returns: a.allocate(n).

[[nodiscard]] static pointer allocate(Alloc& a, size_type n, const_void_pointer hint);  

Returns: a.allocate(n, hint) if that expression is well-formed; otherwise, a.allocate(n).

static void deallocate(Alloc& a, pointer p, size_type n);  

Effects: Calls a.deallocate(p, n).

Throws: Nothing.

template<class T, class... Args>  
static 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 ::new (static_cast<void*>(p)) T(std::forward<Args>(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.

template<class T>  
static void destroy(Alloc& a, T* p);  

Effects: Calls a.destroy(p) if that call is well-formed; otherwise, invokes p->~T().
static 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 Alloc select_on_container_copy_construction(const Alloc& rhs);

Returns: rhs.select_on_container_copy_construction() if that expression is well-formed; other-
wise, rhs.

20.10.10 The default allocator

All specializations of the default allocator satisfy the allocator completeness requirements (16.5.3.5.1).

```cpp
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;
      ~allocator();
      allocator& operator=(const allocator&) = default;
      [[nodiscard]] T* allocate(size_t n);
      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 (6.8.2) 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.

```cpp
[[nodiscard]] T* allocate(size_t n);
```

Returns: A pointer to the initial element of an array of storage of size n * sizeof(T), aligned
appropriately for objects of type T.

Remarks: the storage is obtained by calling ::operator new (17.6.2), but it is unspecified when or
how often this function is called.

Throws: bad_alloc if the storage cannot be obtained.

```cpp
void deallocate(T* p, size_t n);
```

Requires: p shall be a pointer value obtained from allocate(). n shall equal 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

```cpp
template<class T, class U>
bool operator==(const allocator<T>&, const allocator<U>&) noexcept;
```

Returns: true.

```cpp
template<class T, class U>
bool operator!=(const allocator<T>&, const allocator<U>&) noexcept;
```

Returns: false.
20.10.11 Specialized algorithms

Throughout this subclause, the names of template parameters are used to express type requirements for those algorithms defined directly in namespace `std`.

(1.1) If an algorithm’s template parameter is named `InputIterator`, the template argument shall meet the `Cpp17InputIterator` requirements (23.3.5.2).

(1.2) If an algorithm’s template parameter is named `ForwardIterator`, 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, comparison, or indirection through valid iterators.

Unless otherwise specified, if an exception is thrown in the following algorithms, objects constructed by a placement `new-expression` (7.6.2.5) are destroyed in an unspecified order before allowing the exception to propagate.

In the description of the algorithms, operator `-` is used for some of the iterator categories for which it need not be defined. In these cases, the value of the expression `b - a` is the number of increments of `a` needed to make `bool(a == b)` be `true`.

The following additional requirements apply for those algorithms defined in namespace `std::ranges`:

(4.1) The entities defined in the `std::ranges` namespace in this subclause are not found by argument-dependent name lookup (6.4.2). When found by unqualified (6.4.1) name lookup for the postfix-expression in a function call (7.6.1.2), they inhibit argument-dependent name lookup.

(4.2) Overloads of algorithms that take `Range` arguments (24.4.2) behave as if they are implemented by calling `ranges::begin` and `ranges::end` on the `Range`s and dispatching to the overload that takes separate iterator and sentinel arguments.

(4.3) The number and order of deducible template parameters for algorithm declarations is unspecified, except where explicitly stated otherwise. [Note: Consequently, these algorithms may not be called with explicitly-specified template argument lists. — end note]

[Note: When invoked on ranges of potentially overlapping subobjects (6.6.2), the algorithms specified in this subclause 20.10.11 result in undefined behavior. — end note]

Some algorithms defined in this clause make use of the exposition-only function `voidify`:

```cpp
template<class T>
void* voidify(T& obj) noexcept {
    return const_cast<void*>(static_cast<const volatile void*>(addressof(obj)));
}
```

20.10.11.1 Special memory concepts

Some algorithms in this subclause are constrained with the following exposition-only concepts:

```cpp
template<class I>
concept no-throw-input-iterator = // exposition only
    InputIterator<I> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    Same<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 `InputIterator` (23.3.4.9) operations to throw exceptions. — end note]

```cpp
template<class S, class I>
concept no-throw-sentinel = Sentinel<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` (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> &&
   ForwardIterator<I> &&
   no-throw-sentinel<I, I>;

[Note: This concept allows some ForwardIterator (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>>;

20.10.11.2  

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.

20.10.11.3  

template<class ForwardIterator>
void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);

Effects: Equivalent to:

```cpp
for (; first != last; ++first)
   ::new (voidify(*first))
      typename iterator_traits<ForwardIterator>::value_type;
```

namespace ranges {

```cpp
template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
   requires DefaultConstructible<iter_value_t<I>>
   I uninitialized_default_construct(I first, S last);

template<no-throw-forward-range R>
   requires DefaultConstructible<iter_value_t<iterator_t<R>>>>
   safe_iterator_t<R> uninitialized_default_construct(R&& r);

```

}  

Effects: Equivalent to:

```cpp
for (; first != last; ++first)
   ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>;
   return first;
```

template<class ForwardIterator, class Size>
ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);

Effects: Equivalent to:

```cpp
for (; n > 0; (void++)first, --n)
   ::new (voidify(*first))
      typename iterator_traits<ForwardIterator>::value_type;
   return first;
```
namespace ranges {
    template<no-throw-forward-iterator I>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_default_construct_n(I first, iter_difference_t<I> n);
}

Effects: Equivalent to:

return uninitialized_default_construct(counted_iterator(first, n),
    default_sentinel).base();

20.10.11.4 uninitialized_value_construct

template<class ForwardIterator>
void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);

Effects: Equivalent to:

for (; first != last; ++first)
    ::new (voidify(*first))
    typename iterator_traits<ForwardIterator>::value_type();

namespace ranges {
    template<no-throw-forward-iterator I, no-throw-sentinel<I> S>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_value_construct(I first, S last);
    template<no-throw-forward-range R>
    requires DefaultConstructible<iter_value_t<iterator_t<R>>>*
    safe_iterator_t<R> uninitialized_value_construct(R&& r);
}

Effects: Equivalent to:

for (; first != last; ++first)
    ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>();
    return first;

namespace ranges {
    template<no-throw-forward-iterator I>
    requires DefaultConstructible<iter_value_t<I>>
    I uninitialized_value_construct_n(I first, iter_difference_t<I> n);
}

Effects: Equivalent to:

return uninitialized_value_construct(counted_iterator(first, n),
    default_sentinel).base();

20.10.11.5 uninitialized_copy

template<class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
    ForwardIterator result);

Expects: [result, (last - first)) shall not overlap with [first, last).

Effects: Equivalent to:

for (; first != last; ++result, (void) ++first)
    ::new (voidify(*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);

Returns: result.
namespace ranges {
    template<InputIterator I, Sentinel<I> S1, 
        no-throw-forward-iterator O, no-throw-sentinel<O> S2>
        requires Constructible<iter_value_t<O>, iter_reference_t<I>>
        uninitialized_copy_result<I, O>
        uninitialized_copy(I ifirst, S1 ilast, O ofirst, S2 olast);
    template<InputRange IR, no-throw-forward-range OR>
        requires Constructible<iter_value_t<iterator_t<OR>>, iter_reference_t<iterator_t<IR>>> 
        uninitialized_copy_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
        uninitialized_copy(IR&& input_range, OR&& output_range);
}

4 Expects: [ofirst, olast) shall 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 {ifirst, ofirst};

6 template<class InputIterator, class Size, class ForwardIterator>
    ForwardIterator uninitialized_copy_n(InputIterator first, Size n, ForwardIterator result);

6 Expects: [result, n) shall not overlap with [first, n).
7 Effects: Equivalent to:
         for ( ; n > 0; ++result, (void) ++first, --n) {
             ::new (voidify(*result))
             typename iterator_traits<ForwardIterator>::value_type(*first);
         }
8 Returns: result.

namespace ranges {
    template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
        requires Constructible<iter_value_t<O>, iter_reference_t<I>>
        uninitialized_copy_n_result<I, O>
        uninitialized_copy_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
}

9 Expects: [ofirst, olast) shall not overlap with [ifirst, n).
10 Effects: Equivalent to:
         auto t = uninitialized_copy(counted_iterator(ifirst, n),
             default_sentinel, ofirst, olast);
         return {t.in.base(), t.out};

20.10.11.6 uninitialized_move

1 template<class InputIterator, class ForwardIterator>
    ForwardIterator uninitialized_move(InputIterator first, InputIterator last, 
        ForwardIterator result);

1 Expects: [result, (last - first)) shall not overlap with [first, last).
2 Effects: Equivalent to:
         for (; first != last; (void)++result, ++first) {
             ::new (voidify(*result))
             typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
         } return result;

namespace ranges {
    template<InputIterator I, Sentinel<I> S1, 
        no-throw-forward-iterator O, no-throw-sentinel<O> S2>
        requires Constructible<iter_value_t<O>, iter_value_reference_t<I>>
        uninitialized_move_result<I, O>
        uninitialized_move(I ifirst, S1 ilast, O ofirst, S2 olast);
}
template<InputRange IR, no-throw-forward-range OR>
requires Constructible<iter_value_t<iterator_t<OR>>, iter_value_reference_t<iterator_t<IR>>>,
uninitialized_move_result<safe_iterator_t<IR>, safe_iterator_t<OR>>
uninitialized_move(IR&& input_range, OR&& output_range);
}

3 Expects: [ofirst, olast) shall not overlap with [ifirst, ilast).
4 Effects: Equivalent to:
   for (; ifirst != ilast && ofirst != olast; ++ofirst, (void)++ifirst) {
      ::new (voidify(*ofirst))
      remove_reference_t<iter_reference_t<O>>(ranges::iter_move(ifirst));
   }
   return {ifirst, ofirst};
5 [Note: If an exception is thrown, some objects in the range [first, last) are left in a valid, but
   unspecified state. — end note]

template<class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator>
uninitialized_move_n(InputIterator first, Size n, ForwardIterator result);

6 Expects: [result, n) shall not overlap with [first, n).
7 Effects: Equivalent to:
   for (; n > 0; ++result, (void)++first, --n)
      ::new (voidify(*result))
      typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
   return {first, result};

namespace ranges {
   template<InputIterator I, no-throw-forward-iterator O, no-throw-sentinel<O> S>
   requires Constructible<iter_value_t<O>, iter_value_reference_t<I>>
   uninitialized_move_n_result<I, O>
   uninitialized_move_n(I ifirst, iter_difference_t<I> n, O ofirst, S olast);
   }

8 Expects: [ofirst, olast) shall not overlap with [ifirst, n).
9 Effects: Equivalent to:
   auto t = uninitialized_move(counted_iterator(ifirst, n),
                               default_sentinel, ofirst, olast);
   return {t.in.base(), t.out};
10 [Note: If an exception is thrown, some objects in the range [first, n) are left in a valid but unspecified
    state. — end note]

20.10.11.7 uninitialized_fill

template<class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
1 Effects: Equivalent to:
   for (; first != last; ++first)
      ::new (voidify(*first))
      typename iterator_traits<ForwardIterator>::value_type(x);

namespace ranges {
   template<no-throw-forward-iterator I, no-throw-sentinel<I> S, class T>
   requires Constructible<iter_value_t<I>, const T&>
   I uninitialized_fill(I ifirst, S last, const T& x);
   template<no-throw-forward-range R, class T>
   requires Constructible<iter_value_t<iterator_t<R>>, const T&>
   safe_iterator_t<R> uninitialized_fill(R&& r, const T& x);
   }

   Effects: Equivalent to:
for (; first != last; ++first) {
    ::new (voidify(*first)) remove_reference_t<iter_reference_t<I>>(x);
} return first;

template<class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

Effects: Equivalent to:
for (; n--; ++first)
    ::new (voidify(*first))
        typename iterator_traits<ForwardIterator>::value_type(x);
return first;

namespace ranges {
    template<no-throw-forward-iterator I, class T>
    requires Constructible<iter_value_t<I>, const T&>
    I uninitialized_fill_n(I first, iter_difference_t<I> n, const T& x);
}

Effects: Equivalent to:
return uninitialized_fill(counted_iterator(first, n), default_sentinel, x).base();

20.10.11.8 destroy [specialized.destroy]

template<class T>
void destroy_at(T* location);

namespace ranges {
    template<Destructible T>
    void destroy_at(T* location) noexcept;
}

Effects:
(1.1) — If \( T \) is an array type, equivalent to destroy(begin(*location), end(*location)).
(1.2) — Otherwise, equivalent to location->~T().

template<class ForwardIterator>
void destroy(ForwardIterator first, ForwardIterator last);

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>>
    I destroy(I first, S last) noexcept;
    template<no-throw-input-range R>
    requires Destructible<iter_value_t<iterator_t<R>>>
    safe_iterator_t<R> destroy(R&& r) noexcept;
}

Effects: Equivalent to:
for (; first != last; ++first)
    destroy_at(addressof(*first));
return first;

template<class ForwardIterator, class Size>
ForwardIterator destroy_n(ForwardIterator 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>>
    I destroy_n(I first, iter_difference_t<I> n) noexcept;
}

Effects: Equivalent to:
    return destroy(counted_iterator(first, n), default_sentinel).base();

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).
Storage allocated directly with these functions is implicitly declared reachable (see 6.6.5.4.3) 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]

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 (8.7)). 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 Cpp17MoveAssignable, but is not Cpp17CopyConstructible nor Cpp17CopyAssignable. 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]

20.11.1.1 Default deleters
[unique.ptr.dltr]
20.11.1.1 In general
[unique.ptr.dltr.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.
20.11.1.1.2 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;

1 Effects: Constructs a default_delete object from another default_delete<U> object.
2 Remarks: This constructor shall not participate in overload resolution unless U* is implicitly convertible to T*.

void operator()(T* ptr) const;
3 Effects: Calls delete on ptr.
4 Remarks: If T is an incomplete type, the program is ill-formed.

20.11.1.1.3 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;

1 Effects: Constructs a default_delete object from another default_delete<U[]> object.
2 Remarks: This constructor shall not participate in overload resolution unless U(*)[] is convertible to T(*)[].

void operator()(U* ptr) const;
3 Effects: Calls delete[] on ptr.
4 Remarks: If U is an incomplete type, the program is ill-formed. This function shall not participate in overload resolution unless U(*)[] is convertible to T(*)[].

20.11.1.2 unique_ptr for single objects

namespace std {
    template<class T, class D = default_delete<T>> class unique_ptr {
        public:
            using pointer = see below;
            using element_type = T;
            using deleter_type = D;

        // 20.11.1.2.1, 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;

        template<class U, class E>
            unique_ptr(unique_ptr<U, E>&& u) noexcept;

        // 20.11.1.2.2, destructor
            ~unique_ptr();
    };
}
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 \( \text{ptr} \) of type `unique_ptr<T, D>::pointer`, the expression \( d(\text{ptr}) \) is valid and has the effect of disposing of the pointer as appropriate for that deleter.

If the deleter’s type \( D \) is not a reference type, \( D \) shall satisfy the `Cpp17Destructible` requirements (Table 30).

If the qualified-id `remove_reference_t<D>::pointer` is valid and denotes a type (13.9.2), 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 satisfy the `Cpp17NullablePointer` requirements (Table 31).

[Example: Given an allocator type \( X \) (Table 34) and letting \( A \) be a synonym for \( \text{allocator_traits}<X> \), 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

```cpp
constexpr unique_ptr() noexcept;
cconstexpr unique_ptr(nullptr_t) noexcept;
```  
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 \( \text{ptr} \) of type `unique_ptr<T, D>::pointer`, the expression \( d(\text{ptr}) \) 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 satisfy the `Cpp17Destructible` requirements (Table 30).
3 If the qualified-id `remove_reference_t<D>::pointer` is valid and denotes a type (13.9.2), 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 satisfy the `Cpp17NullablePointer` requirements (Table 31).
4 [Example: Given an allocator type \( X \) (Table 34) and letting \( A \) be a synonym for \( \text{allocator_traits}<X> \), 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]

```cpp
explicit unique_ptr(pointer p) noexcept;
```  
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 \( \text{ptr} \) of type `unique_ptr<T, D>::pointer`, the expression \( d(\text{ptr}) \) 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 satisfy the `Cpp17Destructible` requirements (Table 30).
3 If the qualified-id `remove_reference_t<D>::pointer` is valid and denotes a type (13.9.2), 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 satisfy the `Cpp17NullablePointer` requirements (Table 31).
4 [Example: Given an allocator type \( X \) (Table 34) and letting \( A \) be a synonym for \( \text{allocator_traits}<X> \), 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]

```cpp
constexpr unique_ptr() noexcept;
cconstexpr unique_ptr(nullptr_t) noexcept;
```  
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 \( \text{ptr} \) of type `unique_ptr<T, D>::pointer`, the expression \( d(\text{ptr}) \) 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 satisfy the `Cpp17Destructible` requirements (Table 30).
3 If the qualified-id `remove_reference_t<D>::pointer` is valid and denotes a type (13.9.2), 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 satisfy the `Cpp17NullablePointer` requirements (Table 31).
4 [Example: Given an allocator type \( X \) (Table 34) and letting \( A \) be a synonym for \( \text{allocator_traits}<X> \), 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]
deduction (12.3.1.8) would select the function template corresponding to this constructor, then the program is ill-formed.

\texttt{unique\_ptr\langle pointer \ p, \ const \ & \ d \rangle \ noexcept;}
\texttt{unique\_ptr\langle pointer \ p, \ remove\_reference\_t\langle D\rangle \&& \ d \rangle \ noexcept;}

9 \textit{Requires:} For the first constructor, if \( D \) is not a reference type, \( D \) shall satisfy the \texttt{Cpp17CopyConstructible} requirements and such construction shall not exit via an exception. For the second constructor, if \( D \) is not a reference type, \( D \) shall satisfy the \texttt{Cpp17MoveConstructible} requirements and such construction shall not exit via an exception.

10 \textit{Effects:} Constructs a \texttt{unique\_ptr} object which owns \( p \), initializing the stored pointer with \( p \) and initializing the deleter from \texttt{std::forward\langle decltype\langle d\rangle\rangle\langle d\rangle}.

11 \textit{Remarks:} If \( D \) is a reference type, the second constructor is defined as deleted. These constructors shall not participate in overload resolution unless \texttt{is\_constructible\_v\langle D, \ decltype\langle d\rangle\rangle} is \texttt{true}.

12 \textit{Ensures:} \texttt{get() == p, get\_deleter()} returns a reference to the stored deleter. If \( D \) is a reference type then \texttt{get\_deleter()} returns a reference to the lvalue \( d \).

13 \textit{Remarks:} If class template argument deduction (12.3.1.8) would select a function template corresponding to either of these constructors, then the program is ill-formed.

\[\text{Example:}\]
\begin{verbatim}
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<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{verbatim}

\texttt{unique\_ptr\langle unique\_ptr\&\& \ u \rangle \ noexcept;}

15 \textit{Requires:} If \( E \) is not a reference type, \( E \) shall satisfy the \texttt{Cpp17MoveConstructible} requirements (Table 26). Construction of the deleter from an rvalue of type \( D \) shall not throw an exception.

16 \textit{Effects:} Constructs a \texttt{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. \[\text{Note: The construction of the deleter can be implemented with} \texttt{std::forward\langle D\rangle}. \textit{— end note}\]

17 \textit{Ensures:} \texttt{get()} yields the value \( u\texttt{.get()} \) yielded before the construction. \texttt{u.get()} == \texttt{nullptr}. \texttt{get\_deleter()} returns a reference to the stored deleter that was constructed from \texttt{u.get\_deleter()}. If \( D \) is a reference type then \texttt{get\_deleter()} and \texttt{u.get\_deleter()} both reference the same lvalue deleter.

\texttt{template\langle\texttt{class} \ U, \ \texttt{class} \ E\rangle \ \texttt{unique\_ptr\langle unique\_ptr\langle U, E&& \rangle \ u \rangle \ noexcept;}}

18 \textit{Requires:} If \( E \) is not a reference type, construction of the deleter from an rvalue of type \( E \) shall be well-formed and shall not throw an exception. Otherwise, \( E \) is a reference type and construction of the deleter from an lvalue of type \( E \) shall be well-formed and shall not throw an exception.

19 \textit{Remarks:} This constructor shall not participate in overload resolution unless:

19.1 \texttt{unique\_ptr\langle U, E\rangle::pointer} is implicitly convertible to \texttt{pointer},
19.2 \texttt{U} is not an array type, and
19.3 either \( D \) is a reference type and \( E \) is the same type as \( D \), or \( E \) is not a reference type and \( E \) is implicitly convertible to \( D \).

20 \textit{Effects:} Constructs a \texttt{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. \[\text{Note: The deleter constructor can be implemented with} \texttt{std::forward\langle E\rangle}. \textit{— end note}\]

21 \textit{Ensures:} \texttt{get()} yields the value \( u\texttt{.get()} \) yielded before the construction. \texttt{u.get()} == \texttt{nullptr}. \texttt{get\_deleter()} returns a reference to the stored deleter that was constructed from \texttt{u.get\_deleter()}.
20.11.1.2.2 Destructor

~unique_ptr();
1
Requires: The expression get_deleter()(get()) shall be well-formed, shall have well-defined behavior, and shall not throw exceptions. [Note: The use of default_delete requires T to be a complete type. — end note]

Effects: If get() == nullptr there are no effects. Otherwise get_deleter()(get()).

20.11.1.2.3 Assignment

unique_ptr& operator=(unique_ptr&& u) noexcept;
1
Requires: If D is not a reference type, D shall satisfy the Cpp17MoveAssignable requirements (Table 28) and assignment of the deleter from an rvalue of type D shall not throw an exception. Otherwise, D is a reference type; remove_reference_t<D> shall satisfy the Cpp17CopyAssignable requirements and assignment of the deleter from an lvalue of type D shall not throw an exception.

Effects: Calls reset(u.release()) followed by get_deleter() = std::forward<D>(u.get_deleter()).

Returns: *this.

Ensures: u.get() == nullptr.

template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
5
Requires: If E is not a reference type, assignment of the deleter from an rvalue of type E shall be well-formed and shall not throw an exception. Otherwise, E is a reference type and assignment of the deleter from an lvalue of type E shall be well-formed and shall not throw an exception.

Remarks: This operator shall not participate in overload resolution unless:
(6.1) — unique_ptr<U, E>::pointer is implicitly convertible to pointer, and
(6.2) — U is not an array type, and
(6.3) — is_assignable_v<D&, E&&> is true.

Effects: Calls reset(u.release()) followed by get_deleter() = std::forward<E>(u.get_deleter()).

Returns: *this.

Ensures: u.get() == nullptr.

unique_ptr& operator=(nullptr_t) noexcept;
10
Effects: As if by reset().

Ensures: get() == nullptr.

Returns: *this.

20.11.1.2.4 Observers

add_lvalue_reference_t<T> operator*() const;
1
Requires: get() != nullptr.

Returns: *get().

pointer operator->() const noexcept;
3
Requires: get() != nullptr.

Returns: get().

[Note: The use of this function typically requires that T be a complete type. — end note]

pointer get() const noexcept;
6
Returns: The stored pointer.
deleter_type& get_deleter() noexcept;
const deleter_type& get_deleter() const noexcept;

Returns: A reference to the stored deleter.

explicit operator bool() const noexcept;

Returns: get() != nullptr.

20.11.1.2.5 Modifiers

pointer release() noexcept;

Ensures: get() == nullptr.

Returns: The value get() had at the start of the call to release.

void reset(pointer p = pointer()) noexcept;

Requires: The expression get_deleter()(get()) shall be well-formed, shall have well-defined behavior, and shall 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]

Ensures: 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;

Requires: get_deleter() shall be swappable (16.5.3.2) and shall 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;
            unique_ptr& operator=(nullptr_t) noexcept;

        // 20.11.1.3.3, observers
            T& operator[](size_t i) const;
            pointer get() const noexcept;
            deleter_type& get_deleter() noexcept;
            const deleter_type& get_deleter() const noexcept;
            explicit operator bool() const noexcept;

    };

§ 20.11.1.3 623
A specialization for array types is provided with a slightly altered interface.

1. Conversions between different types of `unique_ptr<T[], D>` that would be disallowed for the corresponding pointer-to-array types, and conversions to or from the non-array forms of `unique_ptr`, produce an ill-formed program.

2. Pointers to types derived from `T` are rejected by the constructors, and by `reset`.

3. The observers `operator*` and `operator->` are not provided.

4. The indexing observer `operator[]` is provided.

5. The default deleter will call `delete[]`.

Descriptions are provided below only for members that differ from the primary template.

The template argument `T` shall be a complete type.

### 20.11.1.3.1 Constructors

```cpp
template<class U> explicit unique_ptr(U p) noexcept;
```

This constructor behaves the same as the constructor in the primary template that takes a single parameter of type `pointer` except that it additionally shall not participate in overload resolution unless

1. `U` is the same type as `pointer`, or
2. `pointer` is the same type as `element_type*`, `U` is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.

```cpp
template<class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;
```

This constructor behaves the same as in the primary template, except that it shall not participate in overload resolution unless all of the following conditions hold, where `UP` is `unique_ptr<U, E>`:

1. `U` is an array type, and
2. `pointer` is the same type as `element_type*`, and
3. `UP::pointer` is the same type as `UP::element_type*`, and
4. `UP::element_type(*)[]` is convertible to `element_type(*)[]`, and
5. either `D` is a reference type and `E` is the same type as `D`, or `D` is not a reference type and `E` is implicitly convertible to `D`.

>Note: This replaces the overload-resolution specification of the primary template — end note>
20.11.1.3.2 Assignment

```cpp
template<class U, class E> unique_ptr<
    U, E&> operator=(unique_ptr<
    U, E>&& u)noexcept;
```

This operator behaves the same as in the primary template, except that it shall not participate in overload resolution unless all of the following conditions hold, where UP is `unique_ptr<
    U, E>:`

1. `U` is an array type, and
2. `pointer` is the same type as `element_type*`, and
3. `UP::pointer` is the same type as `UP::element_type*`, and
4. `UP::element_type(*)[]` is convertible to `element_type(*)[]`, and
5. `is_assignable_v<D&, E&&>` is true.

[Note: This replaces the overload-resolution specification of the primary template — end note]

20.11.1.3.3 Observers

```cpp
T& operator[](size_t i) const;
```

Requires: `i <` the number of elements in the array to which the stored pointer points.

Returns: `get()[i].`

20.11.1.3.4 Modifiers

```cpp
void reset(nullptr_t p = nullptr) noexcept;
```

Effects: Equivalent to `reset(pointer()).`

```cpp
template<class U> void reset(U p) noexcept;
```

This function behaves the same as the `reset` member of the primary template, except that it shall not participate in overload resolution unless either

1. `U` is the same type as `pointer`, or
2. `pointer` is the same type as `element_type*`, and `pointer` is a pointer type `V*`, and `V(*)[]` is convertible to `element_type(*)[]`.

20.11.1.4 Creation

```cpp
template<class T, class... Args> unique_ptr<T> make_unique(Args&&... args);
```

Remarks: This function shall not participate in overload resolution unless `T` is not an array.

Returns: `unique_ptr<T>(new T(std::forward<Args>(args)...)).`

```cpp
template<class T> unique_ptr<T> make_unique(size_t n);
```

Remarks: This function shall not participate in overload resolution unless `T` is an array of unknown bound.

Returns: `unique_ptr<T>(new remove_extent_t<T>[n]())`.

```cpp
template<class T, class... Args> unspecified make_unique(Args&&... = delete;
```

Remarks: This function shall not participate in overload resolution unless `T` is an array of known bound.

```cpp
template<class T> unique_ptr<T> make_unique_default_init();
```

Constraints: `T` is not an array.

Returns: `unique_ptr<T>(new T)`.

```cpp
template<class T> unique_ptr<T> make_unique_default_init(size_t n);
```

Constraints: `T` is an array of unknown bound.

Returns: `unique_ptr<T>(new remove_extent_t<T>[n])`.

```cpp
template<class T, class... Args> unspecified make_unique_default_init(Args&&... = delete;
```

Constraints: `T` is an array of known bound.
20.11.1.5 Specialized algorithms

```cpp
template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;
```

Remarks: This function shall not participate in overload resolution unless is_swappable_v<D> is true.

Effects: Calls x.swap(y).

```cpp
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().

```cpp
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().

```cpp
template<class T1, class D1, class T2, class D2>
bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

Requires: Let CT denote
        
        common_type_t<typename unique_ptr<T1, D1>::pointer,
                typename unique_ptr<T2, D2>::pointer>

        Then the specialization less<CT> shall be a function object type (20.14) that induces a strict weak
ordering (25.7) on the pointer values.

Returns: less<CT>()(x.get(), y.get()).

Remarks: If unique_ptr<T1, D1>::pointer is not implicitly convertible to CT or unique_ptr<T2, D2>::pointer
is not implicitly convertible to CT, the program is ill-formed.

```cpp
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.

```cpp
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.

```cpp
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).

```cpp
template<class T, class D>
bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;
```

```cpp
template<class T, class D>
bool operator!=(nullptr_t, const unique_ptr<T, D>& x) noexcept;
```

Returns: (bool)x.

```cpp
template<class T, class D>
bool operator<(const unique_ptr<T, D>& x, nullptr_t);
```

```cpp
template<class T, class D>
bool operator<(nullptr_t, const unique_ptr<T, D>& x);
```

Requires: The specialization less<unique_ptr<T, D>::pointer> shall be a function object type (20.14)
that induces a strict weak ordering (25.7) on the pointer values.

Returns: The first function template returns
        
        less<unique_ptr<T, D>::pointer>()(x.get(), nullptr)

§ 20.11.5
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 !(nullptr < x). The second function template returns !(x < 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);

Effects: Equivalent to: os << p.get();

Returns: os.

Remarks: This function shall not participate in overload resolution unless os << p.get() is a valid expression.

20.11.2 Class bad_weak_ptr

namespace std {
  class bad_weak_ptr : public exception {
    public:
      bad_weak_ptr() noexcept;
      bad_weak_ptr() noexcept;
  };
}

An exception of type bad_weak_ptr is thrown by the shared_ptr constructor taking a weak_ptr.

bad_weak_ptr() noexcept;

Ensures: what() returns an implementation-defined ntbs.

20.11.3 Class template shared_ptr

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 D, class A>
    shared_ptr(nullptr_t p, D d, A a);
template<class Y>
    shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
template<class Y>
    shared_ptr(shared_ptr<Y>&& r, element_type* p) noexcept;
shared_ptr(const shared_ptr<Y>& r) noexcept;
template<class Y>
    shared_ptr(const shared_ptr<Y>& r) noexcept;
shared_ptr(shared_ptr&& r) noexcept;
template<class Y>
    shared_ptr(shared_ptr<Y>&& r) noexcept;

    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);

    // 20.11.3.5, observers
    element_type* get() const noexcept;
    T& operator*() const noexcept;
    T* operator->() const noexcept;
    element_type& operator[](ptrdiff_t i) const;
    long use_count() const noexcept;
    explicit operator bool() 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>
shared_ptr<weak_ptr<T>> -> shared_ptr<T>;
Specializations of \texttt{shared\_ptr} shall be \texttt{Cpp17CopyConstructible}, \texttt{Cpp17CopyAssignable}, and \texttt{Cpp17LessThanComparable}, allowing their use in standard containers. Specializations of \texttt{shared\_ptr} shall be contextually convertible to \texttt{bool}, allowing their use in boolean expressions and declarations in conditions. The template parameter \(T\) of \texttt{shared\_ptr} may be an incomplete type.

\texttt{[Example:]}

\begin{verbatim}
if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
\end{verbatim}

\texttt{—end example}\n
For the purposes of determining the presence of a data race, member functions shall access and modify only the \texttt{shared\_ptr} and \texttt{weak\_ptr} objects themselves and not objects they refer to. Changes in \texttt{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 \textit{compatible with} a pointer type \(T*\) when either \(Y*\) is convertible to \(T*\) or \(Y\) is \texttt{U[N]} and \(T\) is \texttt{cv U[]}.

### 20.11.3.1 Constructors

\texttt{template<class Y> explicit shared\_ptr(Y* p);}  
\texttt{Effects:} Constructs a \texttt{shared\_ptr} object that owns the pointer \(p\). 
\texttt{Ensures:} \texttt{use\_count() == 1 \&\& get() == p}. 

\texttt{template<class Y, class D> shared\_ptr(Y* p, D d);}  
\texttt{Requires:} Construction of \(d\) and a deleter of type \(D\) initialized with \texttt{std::move(d)} shall not throw exceptions. The expression \texttt{d(p)} shall have well-defined behavior and shall not throw exceptions. \(A\) shall satisfy the \texttt{Cpp17Allocator} requirements (Table 34).

\section*{§ 20.11.3.1 629
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.

Ensures: `use_count() == 1 && get() == p`.

Thros: `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

Remarks: When `T` is an array type, this constructor shall not participate in overload resolution unless `is_move_constructible_v<D>` is true, the expression `d(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, this constructor shall not participate in overload resolution unless `is_move_constructible_v<D>` is true, the expression `d(p)` is well-formed, and `Y*` is convertible to `T*`.

```cpp
template<class Y> shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r, element_type* p) noexcept;
```

Effects: Constructs a `shared_ptr` instance that stores `p` and shares ownership with the initial value of `r`.

Ensures: `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]

```cpp
shared_ptr(const shared_ptr& r) noexcept;
template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
```

Remarks: The second constructor shall not participate in overload resolution unless `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`.

Ensures: `get() == r.get() && use_count() == r.use_count()`.

```cpp
shared_ptr(shared_ptr&& r) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
```

Remarks: The second constructor shall not participate in overload resolution unless `Y*` is compatible with `T*`.

Effects: Move constructs a `shared_ptr` instance from `r`.

Ensures: `*this` shall contain the old value of `r`. `r` shall be empty. `r.get() == nullptr`.

```cpp
template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
```

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.

Ensures: `use_count() == r.use_count()`.

Throws: `bad_weak_ptr` when `r.expired()`.

Remarks: This constructor shall not participate in overload resolution unless `Y*` is compatible with `T*`.

```cpp
template<class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
```

Remarks: This constructor shall not participate in overload resolution unless `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. If *this is empty or shares ownership with another shared_ptr instance (use_count() > 1), there are no side effects.
2. Otherwise, if *this owns an object p and a deleter d, d(p) is called.
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;
template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;

Effects: Equivalent to shared_ptr(r).swap(*this).

Returns: *this.

[Note: The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
both assignments may be no-ops. — end note]

shared_ptr& operator=(shared_ptr&& r) noexcept;
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

template<class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

Effects: Equivalent to shared_ptr(std::move(r)).swap(*this).

Returns: *this.

20.11.3.4 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;

2 Requires: get() != 0.

3 Returns: *get().

4 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;

5 Requires: get() != 0.

6 Returns: get().

7 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;

8 Requires: get() != 0 && i >= 0. If T is U[N], i < N.

9 Returns: get()[i].

10 Remarks: When T is not an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

long use_count() const noexcept;

12 Returns: The number of shared_ptr objects, *this included, that share ownership with *this, or 0 when *this is empty.

13 Synchronization: None.

14 [Note: get() == nullptr does not imply a specific return value of use_count(). — end note]

15 [Note: weak_ptr<T>::lock() can affect the return value of use_count(). — end note]

16 [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;

17 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;

18 Returns: An unspecified value such that

(18.1) x.owner_before(y) defines a strict weak ordering as defined in 25.7;

(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_default_init, and allocate_shared_default_init 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_default_init(args);
template<class T, class A, ...>
  shared_ptr<T> allocate_shared_default_init(const A & a, args);

Requires: A shall satisfy the C++17 Allocator requirements (Table 34).

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_default_init 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.

Ensures: 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 l... 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 l... 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, l...) 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_default_init or allocate_shared_default_init, 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.

(7.10) — 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.

(7.11) — 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.

(7.12) — 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>.
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

Returns: A shared_ptr to an object of type T with an initial value T(forward<Args>(args)...).

Remarks: These overloads shall only participate in overload resolution when T is not an array type. The shared_ptr constructors called by these functions enable shared_from_this with the address of the newly constructed object of type T.

[Example:
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> 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[]

Returns: A shared_ptr to an object of type U[N] with a default initial value, where U is remove_extent_t<T>.

Remarks: These overloads shall only participate in overload resolution when T is of the form U[].

[Example:
shared_ptr<double[]> p = make_shared<double[]>(1024); // shared_ptr to a value-initialized double[1024]
shared_ptr<double[1][2][2]> q = make_shared<double[1][2][2]>(6); // shared_ptr to a value-initialized double[6][2][2]
—end example]

template<class T>
shared_ptr<T> make_shared(); // T is U[N]

return: A shared_ptr to an object of type T with a default initial value.

Remarks: These overloads shall only participate in overload resolution when T is of the form U[N].

[Example:
shared_ptr<double[1024]> p = make_shared<double[1024]>();
// shared_ptr to a value-initialized double[1024]
shared_ptr<double[6][2][2]> q = make_shared<double[6][2][2]>();
// shared_ptr to a value-initialized double[6][2][2]
—end example]

template<class T>
shared_ptr<T> make_shared(size_t N,
const remove_extent_t<T>& u); // T is U[]

return: 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.

Remarks: These overloads shall only participate in overload resolution when T is of the form U[].

[Example:
shared_ptr<double[]> p = make_shared<double[]>((1024, 1.0));
§ 20.11.3.6 634
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]

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.

Remarks: These overloads shall only participate in overload resolution when T is of the form U[N].

[Example:
shared_ptr<double[1024]> p = make_shared<double[1024]>(1.0);
// shared_ptr to a double[1024], where each element is 1.0
shared_ptr<double[6][2]> q = make_shared<double[6][2]>({1.0, 0.0});
// shared_ptr to a double[6][2], where each double[2] element is {1.0, 0.0}
// shared_ptr to a vector<int>[4], where each vector has contents {1, 2}
— end example]

template<class T>
shared_ptr<T> make_shared_default_init();

template<class T, class A>
shared_ptr<T> allocate_shared_default_init(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_default_init<X>();
// shared_ptr to a default-initialized X, where each element in X::data has an indeterminate value
shared_ptr<double[1024]> q = make_shared_default_init<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_default_init(size_t N);

template<class T, class A>
shared_ptr<T> allocate_shared_default_init(const A& a, size_t N);

Constraints: T is an array of unknown bound.

Returns: A shared_ptr to an object of type U[N], where U is remove_extent_t<T>.

[Example:
shared_ptr<double[]> p = make_shared_default_init<double[]>(1024);
// shared_ptr to a default-initialized double[1024], where each element has an indeterminate value
— end example]

20.11.3.7 Comparison

template<class T, class U>
bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

Returns: a.get() == b.get().

§ 20.11.3.7 635
template<class T, class U>
    bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;

Returns: 

    less<>()(a.get(), b.get()).

[Note: Defining a comparison function allows shared_ptr objects to be used as keys in associative containers. — end note]

template<class T>
    bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;

template<class T>
    bool operator==(nullptr_t, const shared_ptr<T>& a) noexcept;

Returns: !a.

template<class T>
    bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;

template<class T>
    bool operator<(nullptr_t, const shared_ptr<T>& a) noexcept;

Returns: The first function template returns less<typename shared_ptr<T>::element_type*>()(a.get(), nullptr)

The second function template returns less<typename shared_ptr<T>::element_type*>()(nullptr, a.get())

template<class T>
    bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;

template<class T>
    bool operator>(nullptr_t, const shared_ptr<T>& a) noexcept;

Returns: The first function template returns nullptr < a. The second function template returns a < nullptr.

template<class T>
    bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;

template<class T>
    bool operator!=(nullptr_t, const shared_ptr<T>& a) noexcept;

Returns: The first function template returns !(nullptr < a). The second function template returns !(a < nullptr).

template<class T>
    bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;

template<class T>
    bool operator>=(nullptr_t, const shared_ptr<T>& a) noexcept;

Returns: The first function template returns !(a < nullptr). The second function template returns !(nullptr < a).

20.11.3.8 Specialized algorithms

template<class T>
    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>
    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;

Requires: The expression static_cast<T*>((U*)nullptr) shall be 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. — end note]

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;

Requires: The expression dynamic_cast<T*>((U*)nullptr) shall be well-formed. The expression dynamic_cast<typename shared_ptr<T>::element_type*>(r.get()) shall be well formed and shall have well-defined behavior.
Returns:
(5.1) 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.
(5.2) Otherwise, shared_ptr<T>().

[Note: The seemingly equivalent expression shared_ptr<T>(dynamic_cast<T*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice. — end note]

template<class T, class U>
const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
const_pointer_cast(shared_ptr<U>&& r) noexcept;

Requires: The expression const_cast<T*>((U*)nullptr) shall be well-formed.
Returns:
shared_ptr<T>(R, const_cast<typename shared_ptr<T>::element_type*>(r.get()))
where R is r for the first overload, and std::move(r) for the second.

[Note: The seemingly equivalent expression shared_ptr<T>(const_cast<T*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice. — end note]

template<class T, class U>
reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
reinterpret_pointer_cast(shared_ptr<U>&& r) noexcept;

Requires: The expression reinterpret_cast<T*>((U*)nullptr) shall be well-formed.
Returns:
shared_ptr<T>(R, reinterpret_cast<typename shared_ptr<T>::element_type*>(r.get()))
where R is r for the first overload, and std::move(r) for the second.

[Note: The seemingly equivalent expression shared_ptr<T>(reinterpret_cast<T*>(r.get())) will eventually result in undefined behavior, attempting to delete the same object twice. — end note]

20.11.3.10 get_deleter

template<class D, class T>
D* get_deleter(const shared_ptr<T>&& p) noexcept;

Returns: If p owns a deleter d of type cv-unqualified D, returns addressof(d); otherwise returns nullptr. The returned pointer remains valid as long as there exists a shared_ptr instance that owns d. [Note: It is unspecified whether the pointer remains valid longer than that. This can happen if the
implementation doesn’t destroy the deleter until all weak_ptr instances that share ownership with \( p \) have been destroyed. — end note]}

20.11.3.11 I/O [util.smartptr.shared.io]

```
20.11.4 Class template weak_ptr [util.smartptr.weak]
```

The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```cpp
namespace std {
    template<class T> class weak_ptr {
        public:
            using element_type = remove_extent_t<T>;

            // 20.11.4.1, constructors
            constexpr weak_ptr() noexcept;
            template<class Y>
                weak_ptr(const shared_ptr<Y>& r) noexcept;
            weak_ptr(const weak_ptr& r) noexcept;
            template<class Y>
                weak_ptr(const weak_ptr<Y>& r) noexcept;
            weak_ptr(weak_ptr&& r) noexcept;
            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;
            weak_ptr& operator=(const shared_ptr<Y>& 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;
            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>;
```
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

```cpp
constexpr weak_ptr() noexcept;  // Effects: Constructs an empty `weak_ptr` object.
Ensures: use_count() == 0.

weak_ptr(const weak_ptr& r) noexcept;  // Remarks: The second and third constructors shall not participate in overload resolution unless `Y*` is compatible with `T*`.
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`.
Ensures: use_count() == r.use_count().

weak_ptr(weak_ptr&& r) noexcept;  // Remarks: The second constructor shall not participate in overload resolution unless `Y*` is compatible with `T*`.
Effects: Move constructs a `weak_ptr` instance from `r`.
Ensures: *this shall contain the old value of `r`, `r` shall be empty, `r.use_count() == 0`.
```

### 20.11.4.2 Destructor

```cpp
~weak_ptr();  // Effects: Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.
```

### 20.11.4.3 Assignment

```cpp
weak_ptr& operator=(const weak_ptr& r) noexcept;  // Effects: Equivalent to weak_ptr(r).swap(*this).
template<class Y> weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
template<class Y> weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;

Remarks: The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary object.
Returns: *this.

weak_ptr& operator=(weak_ptr&& r) noexcept;  // Effects: Equivalent to weak_ptr(std::move(r)).swap(*this).
template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;
Returns: *this.
```

### 20.11.4.4 Modifiers

```cpp
void swap(weak_ptr& r) noexcept;  // Effects: Exchanges the contents of `*this` and `r`.

void reset() noexcept;  // 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 25.7;
- 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.

```cpp
namespace std {
    template<class T = void> struct owner_less;

    template<class T> struct owner_less<shared_ptr<T>> {
        bool operator()(const shared_ptr<T>&, const shared_ptr<T>&) const noexcept;
        bool operator()(const shared_ptr<T>&, const weak_ptr<T>&) const noexcept;
        bool operator()(const weak_ptr<T>&, const shared_ptr<T>&) const noexcept;
    };

    template<class T> struct owner_less<weak_ptr<T>> {
        bool operator()(const weak_ptr<T>&, const weak_ptr<T>&) const noexcept;
        bool operator()(const shared_ptr<T>&, const weak_ptr<T>&) const noexcept;
        bool operator()(const weak_ptr<T>&, const shared_ptr<T>&) const noexcept;
    };

    template<> struct owner_less<void> {
        template<class T, class U>
        bool operator()(const shared_ptr<T>&, const shared_ptr<U>&) const noexcept;
        template<class T, class U>
        bool operator()(const shared_ptr<T>&, const weak_ptr<U>&) const noexcept;
        template<class T, class U>
        bool operator()(const weak_ptr<T>&, const shared_ptr<U>&) const noexcept;
        template<class T, class U>
        bool operator()(const weak_ptr<T>&, const weak_ptr<U>&) const noexcept;

        using is_transparent = unspecified;
    };
}
```

\[2\]

operator() shall return \( x.owner_before(y) \). [Note: Note that

\( 2.1 \) — operator() defines a strict weak ordering as defined in 25.7;
under the equivalence relation defined by $\operatorname{operator()}(a, b) \&\& \neg \operatorname{operator()}(b, a)$, two \texttt{shared\_ptr} or \texttt{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 \texttt{enable\_shared\_from\_this<T>} to inherit the \texttt{shared\_from\_this} member functions that obtain a \texttt{shared\_ptr} instance pointing to \texttt{*this}.

Example:

```cpp
struct X: public enable_shared_from_this<X> { }

int main() {
    shared_ptr<X> p(new X);
    shared_ptr<X> q = p->shared_from_this();
    assert(p == q);
    assert(!p.owner_before(q) && !q.owner_before(p)); // p and q share ownership
}
```

— end example

namespace std {
    template<class T> class enable_shared_from_this {
        protected:
            constexpr enable_shared_from_this() noexcept;
            enable_shared_from_this(const enable_shared_from_this&) noexcept;
            enable_shared_from_this& operator=(const enable_shared_from_this&) noexcept;
            ~enable_shared_from_this();

        public:
            shared_ptr<T> shared_from_this();
            shared_ptr<T const> shared_from_this() const;
            weak_ptr<T> weak_from_this() noexcept;
            weak_ptr<T const> weak_from_this() const noexcept;

        private:
            mutable weak_ptr<T> weak_this; // exposition only
    };
}

The template parameter T of \texttt{enable\_shared\_from\_this} may be an incomplete type.

```cpp
constexpr enable_shared_from_this() noexcept;
enable_shared_from_this(const enable_shared_from_this&) 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]
```

```cpp
shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;

Returns: shared_ptr<T>(weak_this).
```

```cpp
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

```cpp
template<class T, class D> struct hash<unique_ptr<T, D>>;
```

Letting \texttt{UP} be \texttt{unique\_ptr<T,D>}, the specialization \texttt{hash<UP>} is enabled (20.14.18) if and only if \texttt{hash<typename UP::pointer>} is enabled. When enabled, for an object \texttt{p} of type \texttt{UP}, \texttt{hash<UP>()(p)}
shall evaluate to the same value as `hash<typename UP::pointer>()(p.get())`. The member functions are not guaranteed to be `noexcept`.

```
template<class T> struct hash<shared_ptr<T>>;
```

For an object `p` of type `shared_ptr<T>`, `hash<shared_ptr<T>>()()(p)` shall evaluate to the same value as `hash<typename shared_ptr<T>::element_type*>(p.get())`.

### 20.11.8 Atomic specializations for smart pointers

The library provides partial specializations of the `atomic` template for shared-ownership smart pointers. The behavior of all operations is as specified in 31.7, 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]

#### 20.11.8.1 Atomic specialization for `shared_ptr`

```
namespace std {
  template<class T> struct atomic<shared_ptr<T>> {
    using value_type = shared_ptr<T>;
    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const noexcept;
    void store(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
    shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
    operator shared_ptr<T>() const 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;
    constexpr atomic() noexcept = default;
    atomic(shared_ptr<T> desired) noexcept;
    atomic(const atomic&) = delete;
    void operator=(const atomic&) = delete;
    void operator=(shared_ptr<T> desired) noexcept;
    private:
    shared_ptr<T> p; // exposition only
  };

  constexpr atomic() noexcept = default;
  atomic(shared_ptr<T> desired) noexcept;
  atomic(const atomic&) = delete;
  void operator=(const atomic&) = delete;
  void operator=(shared_ptr<T> desired) noexcept;
```

**Effects:** Initializes `p{}`.

```
atomic(shared_ptr<T> desired) noexcept;
```

**Effects:** Initializes the object with the value `desired`. Initialization is not an atomic operation (6.8.2). [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(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

**Requires:** The `order` argument shall not be `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

**Effects:** Atomically replaces the value pointed to by this with the value of `desired` as if by `p.swap(desired)`. Memory is affected according to the value of `order`.

```cpp
void operator=(shared_ptr<T> desired) noexcept;
```

**Effects:** Equivalent to `store(desired)`.

```cpp
shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
```

**Requires:** `order` shall not be `memory_order::release` nor `memory_order::acq_rel`.

**Effects:** Memory is affected according to the value of `order`.

**Returns:** Atomically returns `p`.

```cpp
operator shared_ptr<T>() const noexcept;
```

**Effects:** Equivalent to: `return load();`.

```cpp
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 (6.8.2.1).

**Returns:** Atomically returns the value of `p` immediately before the effects.

```cpp
bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order success, memory_order failure) noexcept;
```

**Requires:** `failure` shall not be `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` 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 31.7.1.

If the operation returns `true`, `expected` is not accessed after the atomic update and the operation is an atomic read-modify-write operation (6.8.2) 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(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

**Effects:** Equivalent to:

```cpp
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(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
```

**Effects:** Equivalent to:

```cpp
return compare_exchange_strong(expected, desired, order, fail_order);
```
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.

20.11.8.2 Atomic specialization for weak_ptr

```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;
        void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
        weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
        operator weak_ptr<T>() const noexcept;
        weak_ptr<T> exchange(weak_ptr<T> desired,
            memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order success, memory_order failure) noexcept;
        bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order success, memory_order failure) noexcept;
        bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
            memory_order order = memory_order::seq_cst) noexcept;
        constexpr atomic() noexcept = default;
        atomic(weak_ptr<T> desired) noexcept;
        atomic(const atomic&) = delete;
        void operator=(const atomic&) = delete;
        void operator=(weak_ptr<T> desired) noexcept;

        private:
        weak_ptr<T> p;       // exposition only
    };
}
```

1 Effects: Initializes p{}.

atomic(weak_ptr<T> desired) noexcept;
2 Effects: Initializes the object with the value desired. Initialization is not an atomic operation (6.8.2). [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]

void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
3 Requires: The order argument shall not be 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.

void operator=(weak_ptr<T> desired) noexcept;
5 Effects: Equivalent to store(desired).

weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
6 Requires: order shall not be memory_order::release nor memory_order::acq_rel.
Effects: Memory is affected according to the value of order.

Returns: Atomically returns p.

operator weak_ptr<T>() const noexcept;

Effects: Equivalent to: return load();

weak_ptr<T> exchange(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;

Effects: Atomically replaces p with desired as if by p.swap(desired). Memory is affected according to the value of order. This is an atomic read-modify-write operation (6.8.2.1).

Returns: Atomically returns the value of p immediately before the effects.

bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
memory_order success, memory_order failure) noexcept;

bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
memory_order success, memory_order failure) noexcept;

Requires: failure shall not be memory_order::release nor memory_order::acq_rel.

Effects: If p is equivalent to expected, assigns desired to p and has synchronization semantics corresponding to the value of success, otherwise assigns p to expected and has synchronization semantics corresponding to the value of failure.

Returns: true if p was equivalent to expected, false otherwise.

Remarks: Two weak_ptr objects are equivalent if they store the same pointer value and either share ownership or are both empty. The weak form may fail spuriously. See 31.7.1.

If the operation returns true, expected is not accessed after the atomic update and the operation is an atomic read-modify-write operation (6.8.2) 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(weak_ptr<T>& expected, weak_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(weak_ptr<T>& expected, weak_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.

20.12 Memory resources [mem.res]

20.12.1 Header <memory_resource> synopsis [mem.res.syn]

namespace std::pmr {

// 20.12.2, class memory_resource
class memory_resource;

bool operator==(const memory_resource& a, const memory_resource& b) noexcept;
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;

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

memory_resource class is an abstract interface to an unbounded set of classes encapsulating memory
resources.

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

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

virtual void* do_allocate(size_t bytes, size_t alignment) = 0;

1 Requires: alignment shall be a power of two.
2 Returns: A derived class shall implement this function to return a pointer to allocated storage (6.6.5.4.1) with a size of at least bytes, aligned to the specified alignment.
3 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;

4 Requires: p shall have been returned from a prior call to allocate(bytes, alignment) on a memory resource equal to *this, and the storage at p shall not yet have been deallocated.
5 Effects: A derived class shall implement this function to dispose of allocated storage.
6 Throws: Nothing.

virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;

7 Returns: A derived class shall implement this function to return true if memory allocated from this can be deallocated from other and vice-versa, otherwise false. [Note: 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;

1 Returns: &a == &b || a.is_equal(b).

bool operator!=(const memory_resource& a, const memory_resource& b) noexcept;

2 Returns: !(a == b).

20.12.3 Class template polymorphic_allocator

A specialization of class template pmr::polymorphic_allocator satisfies the Cpp17Allocator requirements (Table 34). 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 satisfy 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& rhs) = delete;

            // 20.12.3.2, member functions
            [[nodiscard]] Tp* allocate(size_t n);
            void deallocate(Tp* p, size_t n);

        } // polymorphic_allocator
    } // namespace std::pmr
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> T* allocate_object(size_t n = 1);
template<class T> void deallocate_object(T* p, size_t n = 1);
template<class T, class... CtorArgs> 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);
   Requires: 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;
   Effects: Sets memory_rsrc to other.resource().

20.12.3.2 Member functions

[[nodiscard]] Tp* allocate(size_t n);
   Effects: Equivalent to:
            return static_cast<Tp*>(memory_rsrc->allocate(n * sizeof(Tp), alignof(Tp)));

void deallocate(Tp* p, size_t n);
   Requires: 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.

void* allocate_bytes(size_t nbytes, size_t alignment = alignof(max_align_t));
   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));
   Effects: Equivalent to memory_rsrc->deallocate(p, nbytes, alignment).

template<class T>
T* allocate_object(size_t n = 1);
   Effects: Allocates memory suitable for holding an array of n objects of type T, as follows:
   — if SIZE_MAX / sizeof(T) < n, throws length_error,
— otherwise equivalent to:

```cpp
return static_cast<T*>(allocate_bytes(n*sizeof(T), alignof(T)));
```

[Note: T is not deduced and must therefore be provided as a template argument. — end note]

```cpp
template<class T>
void deallocate_object(T* p, size_t n = 1);
```

**Effects:** Equivalent to `deallocate_bytes(p, n*sizeof(T), alignof(T))`.

```cpp
template<class T, class CtorArgs...>
T* new_object(CtorArgs&&... ctor_args);
```

**Effects:** Allocates and constructs an object of type T, as follows. Equivalent to:

```cpp
T* p = allocate_object<T>();
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]

```cpp
template<class T>
void delete_object(T* p);
```

**Effects:** Equivalent to:

```cpp
destroy(p);
deleatete_bytes(p);
```

```cpp
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.

```cpp
template<class T>
void destroy(T* p);
```

**Effects:** As if by p->~T().

```cpp
polymorphic_allocator select_on_container_copy_construction() const;
```

**Returns:** `polymorphic_allocator()`.

[Note: The memory resource is not propagated. — end note]

```cpp
memory_resource* resource() const;
```

**Returns:** `memory_resource`.

### 20.12.3.3 Equality

```cpp
template<class T1, class T2>
bool operator==(const polymorphic_allocator<T1>& a,
    const polymorphic_allocator<T2>& b) noexcept;
```

**Returns:** `*a.resource() == *b.resource()`.

```cpp
template<class T1, class T2>
bool operator!=(const polymorphic_allocator<T1>& a,
    const polymorphic_allocator<T2>& b) noexcept;
```

**Returns:** `!(a == b)`. 

§ 20.12.3.3
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().

memory_resource* set_default_resource(memory_resource* r) noexcept;

Effects: If r is non-null, sets the value of the default memory resource pointer to r, otherwise sets the default memory resource pointer to new_delete_resource().

Returns: The previous value of the default memory resource pointer.

Remarks: Calling the set_default_resource and get_default_resource functions shall not incur a data race. A call to the set_default_resource function shall synchronize with subsequent calls to the set_default_resource and get_default_resource functions.

memory_resource* get_default_resource() noexcept;

Returns: The current value of the default memory resource pointer.

20.12.5 Pool resource classes

20.12.5.1 Classes synchronized_pool_resource and unsynchronized_pool_resource

The synchronized_pool_resource and unsynchronized_pool_resource classes (collectively called pool resource classes) are general-purpose memory resources having the following qualities:

(1.1) — 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.

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()) {}
explicit synchronized_pool_resource(memory_resource* upstream)
    : synchronized_pool_resource(pool_options(), upstream) {}
explicit synchronized_pool_resource(const pool_options& opts)
    : synchronized_pool_resource(opts, get_default_resource()) {}
synchronized_pool_resource(const synchronized_pool_resource&) = delete;
virtual ~synchronized_pool_resource();
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);

Requires: 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().

virtual ~synchronized_pool_resource();
virtual ~unsynchronized_pool_resource();

Effects: Calls release().

20.12.5.4 Members

void release();

Effects: Calls upstream_resource()->deallocate() as necessary to release all allocated memory. [Note: The memory is released back to upstream_resource() even if deallocate has not been called for some of the allocated blocks. — end note]

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 (6.6.5.4.1) 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().

§ 20.12.5.4 652
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. A call to deallocate has no effect, thus the amount of memory consumed increases monotonically until the resource is destroyed.
2. The program can supply an initial buffer, which the allocator uses to satisfy memory requests.
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.
4. It is intended for access from one thread of control at a time. Specifically, calls to allocate and deallocate do not synchronize with one another.
5. It frees the allocated memory on destruction, even if deallocate has not been called for some of the allocated blocks.

namespace std::pmr {
    class monotonic_buffer_resource : public memory_resource {
        memory_resource* upstream_rsrc; // exposition only
        void* current_buffer; // exposition only
        size_t next_buffer_size; // exposition only

        public:
            explicit monotonic_buffer_resource(memory_resource* upstream);
            monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);
            monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream);

            monotonic_buffer_resource() : monotonic_buffer_resource(get_default_resource()) {}  // 653
            explicit monotonic_buffer_resource(size_t initial_size) :
                monotonic_buffer_resource(initial_size, get_default_resource()) {}  // 653
            monotonic_buffer_resource(void* buffer, size_t buffer_size) :
                monotonic_buffer_resource(buffer, buffer_size, get_default_resource()) {}  // 653

            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

explicit monotonic_buffer_resource(memory_resource* upstream);
monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);

Requires: upstream shall be the address of a valid memory resource. initial_size, if specified, shall be greater than zero.
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);
```

Requires: `upstream` shall be the address of a valid memory resource. `buffer_size` shall be no larger than the number of bytes in `buffer`.

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();
```

Effects: Calls `release()`.

### 20.12.6.2 Members

```cpp
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]

```cpp
memory_resource* upstream_resource() const;
```

Returns: The value of `upstream_rsrc`.

```cpp
void* do_allocate(size_t bytes, size_t alignment) override;
```

Returns: A pointer to allocated storage (6.6.5.4.1) 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.

```cpp
void do_deallocate(void* p, size_t bytes, size_t alignment) override;
```

Effects: None.

Throws: Nothing.

Remarks: Memory used by this resource increases monotonically until its destruction.

```cpp
bool do_is_equal(const memory_resource& other) const noexcept override;
```

Returns: `this == &other`.

#### 20.13 Class template scoped_allocator_adaptor

```cpp
namespace std {
    // class template scoped allocator adaptor
    template<class OuterAlloc, class... InnerAlloc>
    class scoped_allocator_adaptor;

    // 20.13.5, scoped allocator operators
    template<class OuterA1, class OuterA2, class... InnerAllos>
    bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllos...>& a,
                   const scoped_allocator_adaptor<OuterA2, InnerAllos...>& b) noexcept;

    template<class OuterA1, class OuterA2, class... InnerAllos>
    bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerAllos...>& a,
                   const scoped_allocator_adaptor<OuterA2, InnerAllos...>& b) noexcept;
}
The class template \texttt{scoped\_allocator\_adaptor} is an allocator template that specifies the memory 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 \texttt{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. [\textit{Note: The \texttt{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}]

```cpp
namespace std {
    template<class OuterAlloc, class... InnerAllocs>
    class scoped_allocator_adaptor : public OuterAlloc {
    private:
        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& other) noexcept;
    scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
    scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
    scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;

    template<class OuterA2>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;

    template<class OuterA2>
    scoped_allocator_adaptor(
        scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;

    scoped_allocator_adaptor& operator=(const scoped_allocator_adaptor&) = default;
    scoped_allocator_adaptor& operator=(scoped_allocator_adaptor&) = default;
    ~scoped_allocator_adaptor();
}
```
20.13.2 Member types

using inner_allocator_type = see below;

Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise, scoped_allocator_adaptor<InnerAllocs...>.

using propagate_on_container_copy_assignment = see below;

Type: true_type if allocator_traits<A>::propagate_on_container_copy_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

using propagate_on_container_move_assignment = see below;

Type: true_type if allocator_traits<A>::propagate_on_container_move_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

using propagate_on_container_swap = see below;

Type: true_type if allocator_traits<A>::propagate_on_container_swap::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

using is_always_equal = see below;

Type: true_type if allocator_traits<A>::is_always_equal::value is true for every A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.

20.13.3 Constructors

scoped_allocator_adaptor();

Effects: Value-initializes the OuterAlloc base class and the inner allocator object.

template<class OuterA2>
scoped_allocator_adaptor(OuterA2&& outerAlloc, const InnerAllocs&... innerAllocs) noexcept;

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).

Remarks: This constructor shall not participate in overload resolution unless is_constructible_v<OuterAlloc, OuterA2> is true.

scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;

Effects: Initializes each allocator within the adaptor with the corresponding allocator from other.
scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
Effects: Move constructs each allocator within the adaptor with the corresponding allocator from other.

template<class OuterA2>
scoped_allocator_adaptor(
    const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
Effects: Initializes each allocator within the adaptor with the corresponding allocator from other.
Remarks: This constructor shall not participate in overload resolution unless is_constructible_<v<OuterAlloc, const OuterA2&> is true.

template<class OuterA2>
scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;
Effects: Initializes each allocator within the adaptor with the corresponding rvalue from other.
Remarks: This constructor shall not participate in overload resolution unless is_constructible_<v<OuterAlloc, OuterA2> is true.

20.13.4 Members
In the construct member functions, OUTERMOST(x) is OUTERMOST(x.outer_allocator()) if the expression x.outer_allocator() is valid (13.9.2) 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 [scoped.adaptor.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.outer_allocator() == b.outer_allocator() && a.inner_allocator() == b.inner_allocator()

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: !(a == b).

20.14 Function objects [function.objects]

A function object type is an object type (6.7) that can be the type of the postfix-expression in a function call (7.6.1.2, 12.3.1.1).

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 [functional.syn]

namespace std {
  // 20.14.4, invoke
  template<class F, class... Args>
  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> reference_wrapper<T> ref(T&&) noexcept;
  template<class T> 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> reference_wrapper<T> ref(reference_wrapper<T>) noexcept;
  template<class T> reference_wrapper<const T> cref(reference_wrapper<T>) noexcept;

  template<class T> struct unwrap_reference;
  template<class T> struct unwrap_ref_decay : unwrap_reference<decay_t<T>> {};
  template<class T> using unwrap_ref_decay_t = typename unwrap_ref_decay<T>::type;

  // 20.14.6, arithmetic operations
  template<class T = void> struct plus;
  template<class T = void> struct minus;
  template<class T = void> struct multiplies;

224) Such a type is a function pointer or a class type which has a member \texttt{operator()} or a class type which has a conversion to a pointer to function.
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>;

// 20.14.9, 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> unspecified not_fn(F&& f);

// 20.14.13, function template bind_front
template<class F, class... Args> unspecified bind_front(F&&, Args&&...);

// 20.14.14, bind
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;
template<class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
template<class R, class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);

namespace placeholders {
    // M is the implementation-defined number of placeholders
    see below _1;
    see below _2;

§ 20.14.1
// 20.14.15, member function adaptors
template<class R, class T>
unspecified mem_fn(R T::* noexcept;

// 20.14.16, polymorphic function wrappers
class bad_function_call;

template<class R, class... ArgTypes> class function; // not defined

template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

template<class R, class... ArgTypes>
void swap(function<R(ArgTypes)...>*, function<R(ArgTypes)...>*) noexcept;

template<class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>*, nullptr_t) noexcept;

template<class R, class... ArgTypes>
bool operator!=(const function<R(ArgTypes...)>*, nullptr_t) noexcept;

template<class RandomAccessIterator,
class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
class BinaryPredicate = equal_to>>
struct default_searcher;

template<class RandomAccessIterator,
class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
class BinaryPredicate = equal_to>>
struct boyer_moore_searcher;

template<class RandomAccessIterator,
class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
class BinaryPredicate = equal_to>>
struct boyer_moore_horspool_searcher;

// 20.14.17, searchers
template<class ForwardIterator, class BinaryPredicate = equal_to>
class default_searcher;

// 20.14.18, hash function primary template
template<class T>
struct hash;

// 20.14.14, function object binders
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;

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 \texttt{double} and put the result into \( a \), it can do:
\[
\text{transform}(a.\text{begin}(), a.\text{end}(), b.\text{begin}(), a.\text{begin}(), \text{plus}\langle\text{double}\rangle());
\]
—end example]

[Example: To negate every element of \( a \):
\[
\text{transform}(a.\text{begin}(), a.\text{end}(), a.\text{begin}(), \text{negate}\langle\text{double}\rangle());
\]
—end example]

### 20.14.2 Definitions

The following definitions apply to this Clause:

1. A \textit{call signature} is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

2. A \textit{callable type} is a function object type (20.14) or a pointer to member.

3. A \textit{callable object} is an object of a callable type.

4. A \textit{call wrapper type} is a type that holds a callable object and supports a call operation that forwards to that object.

5. A \textit{call wrapper} is an object of a call wrapper type.

6. A \textit{target object} is the callable object held by a call wrapper.

7. A \textit{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 \textit{bound argument entities}.

8. The target object and bound argument entities of the call wrapper are collectively referred to as \textit{state entities}.

### 20.14.3 Requirements

1. Define \texttt{INVOKE}(f, t_1, t_2, \ldots, t_N) as follows:
   
   \begin{enumerate}[\itemsep=0em]
   \item \((t_1.\text{\*f})(t_2, \ldots, t_N)\) when \( f \) is a pointer to a member function of a class \( T \) and \texttt{is_base_of_v<T, remove Reference T<decltype(t_1)>>} is \texttt{true};
   \item \((t_1.\text{get()}.\text{\*f})(t_2, \ldots, t_N)\) when \( f \) is a pointer to a member function of a class \( T \) and \texttt{remove Cvref T<decltype(t_1)>>} is a specialization of \texttt{reference wrapper};
   \item \(((\*t_1).\text{\*f})(t_2, \ldots, t_N)\) when \( f \) is a pointer to a member function of a class \( T \) and \( t_1 \) does not satisfy the previous two items;
   \item \( t_1.\text{\*f} \) when \( N == 1 \) and \( f \) is a pointer to data member of a class \( T \) and \texttt{is_base_of_v<T, remove Reference T<decltype(t_1)>>} is \texttt{true};
   \item \( t_1.\text{get()}.\text{\*f} \) when \( N == 1 \) and \( f \) is a pointer to data member of a class \( T \) and \texttt{remove Cvref T<decltype(t_1)>>} is a specialization of \texttt{reference wrapper};
   \item \(((\*t_1).\text{\*f})(t_2, \ldots, t_N)\) when \( N == 1 \) and \( f \) is a pointer to data member of a class \( T \) and \( t_1 \) does not satisfy the previous two items;
   \item \( f(t_1, t_2, \ldots, t_N) \) in all other cases.
   \end{enumerate}

2. Define \texttt{INVOKE<R>\langle f, t_1, t_2, \ldots, t_N \rangle} as \texttt{static_cast<\texttt{void}}\rangle\langle\texttt{INVOKE}(f, t_1, t_2, \ldots, t_N)\rangle\rangle if \texttt{R} is a cv void, otherwise \texttt{INVOKE}(f, t_1, t_2, \ldots, t_N) implicitly converted to \texttt{R}.

3. Every call wrapper (20.14.2) is \texttt{Cpp17MoveConstructible}. A \textit{argument forwarding call wrapper} is a call wrapper that can be called with an arbitrary argument list and delivers the arguments to the wrapped callable object as arguments. This forwarding step delivers rvalue arguments as rvalue references and lvalue arguments as lvalue references. A \textit{simple call wrapper} is an argument forwarding call wrapper that is \texttt{Cpp17CopyConstructible} and \texttt{Cpp17CopyAssignible} and whose copy constructor, move constructor, copy assignment operator, and move assignment operator do not throw exceptions. [Note: In a typical implementation, argument forwarding call wrappers have an overloaded function call operator of the form
\[
\text{template<class... UnBoundArgs>}
\text{R operator()\langle UnBoundArgs&&... unbound_args \rangle cu-qual;}
\]
—end note]
A **perfect forwarding call wrapper** is an argument forwarding call wrapper that forwards its state entities to the underlying call expression. This forwarding step delivers a state entity of type \( \text{cv} \ T \& \) when the call is performed on an lvalue of the call wrapper type and as \( \text{cv} \ T \&& \) otherwise, where \( \text{cv} \) represents the cv-qualifiers of the call wrapper and where \( \text{cv} \) shall be neither `volatile` nor `const volatile`.

A **call pattern** defines the semantics of invoking a perfect forwarding call wrapper. A postfix call performed on a perfect forwarding call wrapper is expression-equivalent (16.3.11) to an expression \( e \) determined from its call pattern \( \text{cp} \) by replacing all occurrences of the arguments of the call wrapper and its state entities with references as described in the corresponding forwarding steps.

The copy/move constructor of a perfect forwarding call wrapper has the same apparent semantics as if memberwise copy/move of its state entities were performed (11.3.4.2). [Note: This implies that each of the copy/move constructors has the same exception-specification as the corresponding implicit definition and is declared as `constexpr` if the corresponding implicit definition would be considered to be `constexpr`. — end note]

Perfect forwarding call wrappers returned by a given standard library function template have the same type if the types of their corresponding state entities are the same.

### 20.14.4 Function template *invoke* [func.invoke]

```cpp
template<class F, class... Args>
invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
  noexcept(is_nothrow_invocable_v<F, Args...>);
```

1. **Returns:** \( \text{INVOKE} \text{(std::forward<F>(f), std::forward<Args>(args)...) (20.14.3).} \)

### 20.14.5 Class template *reference_wrapper* [refwrap]

```cpp
namespace std {
  template<class T> class reference_wrapper {
    public:
      // types
      using type = T;

      // construct/copy/destroy
      template<class U>
      reference_wrapper(U&&) noexcept(see below);
      reference_wrapper(const reference_wrapper& x) noexcept;

      // assignment
      reference_wrapper& operator=(const reference_wrapper& x) noexcept;

      // access
      operator T& () const noexcept;
      T& get() const noexcept;

      // invocation
      template<class... ArgTypes>
      invoke_result_t<T&, ArgTypes...> operator()(ArgTypes&&...) const;
    template<class T>
    reference_wrapper(T&) -> reference_wrapper<T>;
  }
```

1. **reference_wrapper<T>** is a `Cpp17CopyConstructible` and `Cpp17CopyAssignable` wrapper around a reference to an object or function of type \( T \).
2. **reference_wrapper<T>** is a trivially copyable type (6.7).
3. The template parameter \( T \) of `reference_wrapper` may be an incomplete type.

#### 20.14.5.1 Constructors and destructor [refwrap.const]

```cpp
template<class U>
reference_wrapper(U&& u) noexcept(see below);
```

1. **Remarks:** Let \( \text{FUN} \) denote the exposition-only functions.
This constructor shall not participate in overload resolution unless the expression `FUN(declval<U>())` is well-formed and `is_same_v<remove_cvref_t<U>, reference_wrapper>` is false. The expression inside `noexcept` is equivalent to `noexcept(FUN(declval<U>())`).

**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`.

```
reference_wrapper(const reference_wrapper& x) noexcept;
```

**Effects:** Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

### 20.14.5.2 Assignment

```
reference_wrapper& operator=(const reference_wrapper& x) noexcept;
```

**Ensures:** `*this` stores a reference to `x.get()`.

### 20.14.5.3 Access

```
operator T& () const noexcept;
T& get() const noexcept;
```

**Returns:** The stored reference.

### 20.14.5.4 Invocation

```
template<class... ArgTypes>
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

```
template<class T> reference_wrapper<T> ref(T& t) noexcept;
template<class T> reference_wrapper<T> ref(reference_wrapper<T> t) noexcept;
template<class T> reference_wrapper<const T> cref(const T& t) noexcept;
template<class T> reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;
```

**Returns:**

1. `reference_wrapper<T>(t)`.
2. `ref(t.get())`.
3. `reference_wrapper<const T>(t)`.
4. `cref(t.get())`.
5. `cref(t.get())`.

### 20.14.5.6 Transformation type trait `unwrap_reference`

```
template<class T>
struct unwrap_reference;
```

**If** `T` is a specialization `reference_wrapper<X>` for some type `X`, the member typedef `type` of `unwrap_reference<T>` is `X&`, otherwise it is `T`.

### 20.14.6 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
template<class T = void> struct plus {
    constexpr T operator()(const T& x, const T& y) const;
};

cconstexpr 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
template<class T = void> struct minus {
    constexpr T operator()(const T& x, const T& y) const;
};

cconstexpr 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;
};

cconstexpr T operator()(const T& x, const T& y) const;
Returns: x * y.

template<> struct multiplies<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) * std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) * std::forward<U>(u));
Returns: std::forward<T>(t) * std::forward<U>(u).

20.14.6.4 Class template divides
template<class T = void> struct divides {
    constexpr T operator()(const T& x, const T& y) const;
};

cconstexpr T operator()(const T& x, const T& y) const;
```cpp
};

constexpr T operator() (const T& x, const T& y) const;
1
Returns: x / y.

template<> struct divides<void> {
  template<class T, class U> constexpr auto operator() (T&& t, U&& u) const
  -> decltype(std::forward<T>(t) / std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator() (T&& t, U&& u) const
-> decltype(std::forward<T>(t) / std::forward<U>(u));
2
Returns: std::forward<T>(t) / std::forward<U>(u).

20.14.6.5 Class template modulus

[arithmetic.operations.modulus]

template<class T = void> struct modulus {
  constexpr T operator() (const T& x, const T& y) const;
};

castexpr T operator() (const T& x, const T& y) const;
1
Returns: x % y.

template<> struct modulus<void> {
  template<class T, class U> constexpr auto operator() (T&& t, U&& u) const
  -> decltype(std::forward<T>(t) % std::forward<U>(u));

  using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator() (T&& t, U&& u) const
-> decltype(std::forward<T>(t) % std::forward<U>(u));
2
Returns: std::forward<T>(t) % std::forward<U>(u).

20.14.6.6 Class template negate

[arithmetic.operations.negate]

template<class T = void> struct negate {
  constexpr T operator() (const T& x) const;
};

castexpr T operator() (const T& x) const;
1
Returns: -x.

template<> struct negate<void> {
  template<class T> constexpr auto operator() (T&& t) const
  -> decltype(-std::forward<T>(t));

  using is_transparent = unspecified;
};

template<class T> constexpr auto operator() (T&& t) const
-> decltype(-std::forward<T>(t));
2
Returns: -std::forward<T>(t).

20.14.7 Comparisons

[comparisons]
1
The library provides basic function object classes for all of the comparison operators in the language (7.6.9, 7.6.10).
2
For templates less, greater, less_equal, and greater_equal, the specializations for any pointer type yield a strict total order that is consistent among those specializations and is also consistent with the partial order imposed by the built-in operators <, >, <=, >=. \[Note: When a < b is well-defined for pointers a and b of type P, this implies (a < b) == less<P>()(a, b), (a > b) == greater<P>()(a, b), and so\]
forth. — end note] For template specializations `less<void>`, `greater<void>`, `less_equal<void>`, and `greater_equal<void>`, if the call operator calls a built-in operator comparing pointers, the call operator yields a strict total order that is consistent among those specializations and is also consistent with the partial order imposed by those built-in operators.

### 20.14.7.1 Class template `equal_to`

```
20.14.7.1 Class template `equal_to`      
[comparisons.equal.to]

template<class T = void> struct equal_to {
    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: `std::forward<T>(t) == std::forward<U>(u)`.

### 20.14.7.2 Class template `not_equal_to`

```
20.14.7.2 Class template `not_equal_to`      
[comparisons.not.equal.to]

template<class T = void> struct not_equal_to {
    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: `std::forward<T>(t) != std::forward<U>(u)`.

### 20.14.7.3 Class template `greater`

```
20.14.7.3 Class template `greater`      
[comparisons.greater]

template<class T = void> struct greater {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

Returns: `x > y`.

```
template<> struct greater<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) > std::forward<U>(u));
};
```

Using `is_transparent = unspecified`.

```
template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) > std::forward<U>(u));
```

Returns: `std::forward<T>(t) > std::forward<U>(u)`.

§ 20.14.7.3

666
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;
};
```

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;
};
```

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;
};
```

Returns: \( x \geq y \).

```cpp
template<> struct greater_equal<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) \geq\ std::forward<U>(u));

    using is_transparent = unspecified;
};
```

20.14.7.6 Class template less_equal
[comparisons.less.equal]

```cpp
template<class T = void> struct less_equal {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

Returns: \( x \leq y \).

```cpp
template<> struct less_equal<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) \leq\ std::forward<U>(u));

    using is_transparent = unspecified;
};
```

20.14.8 Concept-constrained comparisons
[range cmp]

In this subclause, \texttt{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. \texttt{BUILTIN_PTR_CMP(T, op, U)} is true if and only if \( op \) in the expression \texttt{declval<T>() op declval<U>()} resolves to a built-in operator comparing pointers.
There is an implementation-defined strict total ordering over all pointer values of a given type. This total ordering is consistent with the partial order imposed by the builtin operators \(<\), \(\leq\), and \(\geq\).

```cpp
template<class T, class U>
requires EqualityComparableWith<T, U> || BUILTIN_PTR_CMP(T, ==, U)
constexpr bool operator()(T&& t, U&& u) const;
```

---

3

`Expects:` 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 \(P\) shall be 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 `false` if either (the converted value of) \(t\) precedes \(u\) or \(u\) precedes \(t\) in the implementation-defined strict total order over pointers of type \(P\) and otherwise `true`.

3.2

- Otherwise, equivalent to: `return std::forward<T>(t) == std::forward<U>(u);`

---

5

`operator()` has effects equivalent to:

```cpp
return !ranges::equal_to{}(std::forward<T>(t), std::forward<U>(u));
```

---

6

`Expects:` 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 \(P\) shall be 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)` shall be `true`.

`Effects:`

6.1

- 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 `true` if (the converted value of) \(t\) precedes \(u\) in the implementation-defined strict total order over pointers of type \(P\) and otherwise `false`.

6.2

- Otherwise, equivalent to: `return std::forward<T>(t) < std::forward<U>(u);`
struct ranges::greater_equal {
    template<class T, class U>
        requires StrictTotallyOrderedWith<T, U> || BUILTIN_PTR_CMP(T, <, U)
    constexpr bool operator()(T&& t, U&& u) const;
    using is_transparent = unspecified;
};

operator() has effects equivalent to:
    return !ranges::less{}(std::forward<T>(t), std::forward<U>(u));

struct ranges::less_equal {
    template<class T, class U>
        requires StrictTotallyOrderedWith<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

```cpp
template<class T = void> struct logical_and {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

Returns: \(x \&\& y\).

```cpp
template<> struct logical_and<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) \&\& std::forward<U>(u));
    using is_transparent = unspecified;
};
```

Returns: \(std::forward<T>(t) \&\& std::forward<U>(u)\).

### 20.14.9.2 Class template logical_or

```cpp
template<class T = void> struct logical_or {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

Returns: \(x \|\| y\).

```cpp
template<> struct logical_or<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) \|\| std::forward<U>(u));
    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).

20.14.9.3 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;

1 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));

2 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;

1 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));

2 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;

1 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));

2 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

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

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

template<class F> unspecified not_fn(F&& f);

1 In the text that follows:
g is a value of the result of a not_fn invocation,
— FD is the type decay_t<F>,
— fd is the target object of g (20.14.2) of type FD, initialized with the initializer (std::forward<F>(f)) (9.3),
— call_args is an argument pack used in a function call expression (7.6.1.2) of g.

Mandates: is_constructible_v<FD, F> && is_move_constructible_v<FD> is true.
Expects: FD meets the Cpp17MoveConstructible requirements.
Returns: A perfect forwarding call wrapper g with call pattern !invoke(fd, call_args...).
Throws: Any exception thrown by the initialization of fd.

20.14.13 Function template bind_front

template<class F, class... Args>
unspecifted bind_front(F&& f, Args&&... args);

In the text that follows:
— g is a value of the result of a bind_front invocation,
— FD is the type decay_t<F>,
— fd is the target object of g (20.14.2) of type FD initialized with the initializer (std::forward<F>(f)) (9.3),
— BoundArgs is a pack that denotes std::unwrap_ref_decay_t<Args>..., 
— bound_args is a pack of bound argument entities of g (20.14.2) of types BoundArgs..., initialized with initializers (std::forward<Args>(args))..., respectively, and
— call_args is an argument pack used in a function call expression (7.6.1.2) of g.

Mandates:

conjunction_v<is_constructible<FD, F>, is_move_constructible<FD>,
is_constructible<BoundArgs, Args>..., is_move_constructible<BoundArgs>...> is true.
Expects: FD meets the Cpp17MoveConstructible requirements. For each Ti in BoundArgs, if Ti is an object type, Ti 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

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.

Instantiations of the is_bind_expression template shall satisfy the Cpp17UnaryTypeTrait requirements (20.15.1). The implementation shall provide a definition that has a base characteristic of true_type if T is a type returned from bind, otherwise it shall have 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

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.

Instantiations of the `is_placeholder` template shall satisfy the Cpp17UnaryTypeTrait requirements (20.15.1). The implementation shall provide a definition that has the base characteristic of `integral_constant<int, J>` if `T` is the type of `std::placeholders::_J`, otherwise it shall have 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.

### Function template `bind`

In the text that follows:

- `FD` is the type `decay_t<F>`,
- `fd` is an lvalue of type `FD` constructed from `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>`;
- `td` is an lvalue of type `TD_i` constructed from `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>
unspec_decl bind(F&& f, BoundArgs&&... bound_args);
```

**Requires:** `is_constructible_v<FD, F>` shall be true. For each `T_i` in `BoundArgs`, `is_constructible_v<decay_t<TD_i>, T_i>` shall be true. `INVOKE(fd, w_1, w_2, ..., w_N)` (20.14.3) shall be a valid expression for some values `w_1, w_2, ..., w_N`, where `N` has the value `sizeof...(bound_args)`. The cv-qualities `cv` of the call wrapper `g`, as specified below, shall be neither `volatile` nor `const` `volatile`.

**Returns:** An argument forwarding call wrapper `g` (20.14.3). The effect of `g(u_1, u_2, ..., u_M)` shall be

```cpp
INVOKE(fd, std::forward<V_1>(v_1), std::forward<V_2>(v_2), ..., std::forward<V_N>(v_N))
```

where the values and types of the bound arguments `v_1, v_2, ..., v_N` are determined as specified below. The copy constructor and move constructor of the argument forwarding call wrapper shall throw an exception if and only if the corresponding constructor of `FD` or of any of the types `TD_i` throws an exception.

**Throws:** Nothing unless the construction of `fd` or of one of the values `td_i` throws an exception.

**Remarks:** The return type shall satisfy the Cpp17MoveConstructible requirements. If all of `FD` and `TD_i` satisfy the Cpp17CopyConstructible requirements, then the return type shall satisfy the Cpp17CopyConstructible requirements. [Note: This implies that all of `FD` and `TD_i` are Cpp17MoveConstructible. — end note]

```cpp
template<class R, class F, class... BoundArgs>
unspec_decl bind(R&& r, F&& f, BoundArgs&&... bound_args);
```

**Requires:** `is_constructible_v<FD, F>` shall be true. For each `T_i` in `BoundArgs`, `is_constructible_v<decay_t<TD_i>, T_i>` shall be true. `INVOKE(fd, w_1, w_2, ..., w_N)` shall be a valid expression for some values `w_1, w_2, ..., w_N`, where `N` has the value `sizeof...(bound_args)`. The cv-qualities `cv` of the call wrapper `g`, as specified below, shall be neither `volatile` nor `const` `volatile`.

**Returns:** An argument forwarding call wrapper `g` (20.14.3). The effect of `g(u_1, u_2, ..., u_M)` shall be

```cpp
INVOKE<R>(fd, std::forward<V_1>(v_1), std::forward<V_2>(v_2), ..., std::forward<V_N>(v_N))
```

where the values and types of the bound arguments `v_1, v_2, ..., v_N` are determined as specified below. The copy constructor and move constructor of the argument forwarding call wrapper shall throw an exception if and only if the corresponding constructor of `FD` or of any of the types `TD_i` throws an exception.
Throws: Nothing unless the construction of \( \text{fd} \) or of one of the values \( \text{td}_i \) throws an exception.

Remarks: The return type shall satisfy the \emph{Cpp17MoveConstructible} requirements. If all of \( \text{FD} \) and \( \text{TD}_i \) satisfy the \emph{Cpp17CopyConstructible} requirements, then the return type shall satisfy the \emph{Cpp17CopyConstructible} requirements. [Note: This implies that all of \( \text{FD} \) and \( \text{TD}_i \) are \emph{Cpp17MoveConstructible}. — end note]}

The values of the bound arguments \( v_1, v_2, \ldots, v_N \) and their corresponding types \( V_1, V_2, \ldots, V_N \) depend on the types \( \text{TD}_i \) derived from the call to \text{bind} and the cv-qualifiers \( \text{cv} \) of the call wrapper \( g \) as follows:

1. if \( \text{TD}_i \) is \text{reference_wrapper}\( \langle T \rangle \), the argument is \( \text{td}_i(). \) and its type \( V \) is \( T & \);
2. if the value of \( \text{is\_bind\_expression}\langle \langle \text{TD}_i \rangle \rangle \) is true, the argument is \( \text{td}_i(\text{std::forward}\langle U_j \rangle\langle u_j \rangle \ldots) \) and its type \( V \) is \( \text{invoke\_result\_t}\langle \langle \text{TD}_i \rangle, \text{cv} \ k, U_j \ldots \rangle & \);
3. if the value \( j \) of \( \text{is\_placeholder}\langle \langle \text{TD}_i \rangle \rangle \) is not zero, the argument is \( \text{std::forward}\langle U_j \rangle\langle u_j \rangle \) and its type \( V \) is \( U_j \ & \);
4. otherwise, the value is \( \text{td}_i \) and its type \( V \) is \( \text{TD}_i \ & \).

20.14.14 Placeholders

namespace std::placeholders {
  // \( M \) is the implementation-defined number of placeholders
  see below _1;
  see below _2;
  .
  .
  see below _M;
}

All placeholder types shall be \emph{Cpp17DefaultConstructible} and \emph{Cpp17CopyConstructible}, and their default constructors and copy/move constructors shall not throw exceptions. It is implementation-defined whether placeholder types are \emph{Cpp17CopyAssignible}. \emph{Cpp17CopyAssignible} placeholders' copy assignment operators shall not throw exceptions.

Placeholders should be defined as:

```cpp
inline constexpr unspecified _1{};
```

If they are not, they shall be declared as:

```cpp
extern unspecified _1;
```

20.14.15 Function template \text{mem\_fn}

```cpp
template<class R, class T>
unspecified mem\_fn(R T::* pm) noexcept;
```

Returns: A simple call wrapper (20.14.2) \( \text{fn} \) such that the expression \( \text{fn}(t, a_2, \ldots, a_N) \) is equivalent to \( \text{INVOKE}(pm, t, a_2, \ldots, a_N) \) (20.14.3).

20.14.16 Polymorphic function wrappers

This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.

20.14.16.1 Class \text{bad\_function\_call}

An exception of type \text{bad\_function\_call} is thrown by \text{function\::operator()} (20.14.16.2.4) when the function wrapper object has no target.

```cpp
namespace std {
  class bad\_function\_call : public exception {
    public:
      // 20.14.16.1.1, constructor
      bad\_function\_call() noexcept;
  };
}
```

20.14.16.1.1 Constructor

```cpp
bad\_function\_call() noexcept;
```

Effects: Constructs a \text{bad\_function\_call} object.
Ensures: `what()` returns an implementation-defined `intbs`.

### 20.14.16.2 Class template function

```cpp
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&);
            function(function&&) noexcept;
            template<class F> function(F);
            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 comparisons
        template<class R, class... ArgTypes>
        bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

        template<class R, class... ArgTypes>
        bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

        template<class R, class... ArgTypes>
        bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

        template<class R, class... ArgTypes>
        bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

        // 20.14.16.2.7, specialized algorithms
        template<class R, class... ArgTypes>
        void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;
    }

    template<class R, class... ArgTypes>
    function(R(*)(ArgTypes...)) -> function<R(ArgTypes...)>

    template<class F> function(F) -> function<see below>;
```
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.

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).

The `function` class template is a call wrapper (20.14.2) whose call signature (20.14.2) is `R(ArgTypes...)`.

[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

#### Constructors

1. `function() noexcept;`  
   **Ensures:** `!*this`.

2. `function(nullptr_t) noexcept;`  
   **Ensures:** `!*this`.

3. `function(const function& f);`  
   **Ensures:** `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

4. `function(function&& f) noexcept;`  
   **Ensures:** If `!f`, `*this` has no target; otherwise, `*this` targets a copy of `f.target()`.

5. `template<class F> function(F f);`  
   **Requires:** `F` shall be `Cpp17CopyConstructible`.

6. `template<class F> function(F) -> function<see below>;`  
   **Remarks:** This deduction guide participates in overload resolution only if `&F::operator()` is well-formed when treated as an unevaluated operand. In that case, if `decltype(&F::operator())` is of the form `R(G::*)(A...) cv l_opt noexcept opt` for a class type `G`, then the deduced type is `function<R(A...)>>`.

[Example:]

```cpp
void f() {
  int i{5};
  ...}
```
function g = \[k\](double) { return i; }; // deduces function<int(double)>

—end example

function& operator=(const function& f);  
Effects: As if by function(f).swap(*this);
Returns: *this.

function& operator=(function&& f);  
Effects: Replaces the target of *this with the target of f.
Returns: *this.

function& operator=(nullptr_t) noexcept;  
Effects: If *this != nullptr, destroys the target of this.
Ensures: !(*this).
Returns: *this.

template<class F> function& operator=(F&& f);  
Effects: As if by:
function(std::forward<F>(f)).swap(*this);
Returns: *this.
Remarks: This assignment operator shall not participate in overload resolution unless decay_t<F> is Lvalue-Callable (20.14.16.2) for argument types ArgTypes... and return type R.

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;  
Effects: 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: INVOKE<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;

template<class R, class... ArgTypes>
bool operator==(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;

Returns: !f.

template<class R, class... ArgTypes>
bool operator!=(const function<R(ArgTypes...)>& f, nullptr_t) noexcept;

template<class R, class... ArgTypes>
bool operator!=(nullptr_t, const function<R(ArgTypes...)>& f) noexcept;

Returns: (bool)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 satisfy the Cpp17CopyConstructible and Cpp17CopyAssignable requirements. Template parameters named

(2.1) — ForwardIterator,
(2.2) — ForwardIterator1,
(2.3) — ForwardIterator2,
(2.4) — RandomAccessIterator,
(2.5) — RandomAccessIterator1,
(2.6) — RandomAccessIterator2, and
(2.7) — BinaryPredicate

of templates specified in this subclause 20.14.17 shall satisfy the same requirements and semantics as specified in 25.1. Template parameters named Hash shall satisfy the Cpp17Hash requirements (Table 32).

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

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
};
constexpr default_searcher(ForwardIterator pat_first, ForwardIterator pat_last,
    BinaryPredicate pred = BinaryPredicate());

Effects: Constructs a default_searcher object, initializing pat_first_ with pat_first, pat_last_ with pat_last, and pred_ with pred.

Throws: Any exception thrown by the copy constructor of BinaryPredicate or ForwardIterator1.

template<class ForwardIterator2>
constexpr pair<ForwardIterator2, ForwardIterator2>
operator()(ForwardIterator2 first, ForwardIterator2 last) const;

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 Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>,
    class BinaryPredicate = equal_to<>>
class boyer_moore_searcher {
public:
    boyer_moore_searcher(RandomAccessIterator1 pat_first,
    RandomAccessIterator1 pat_last,
    Hash hf = Hash(),
    BinaryPredicate pred = BinaryPredicate());

    template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
    operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

private:
    RandomAccessIterator1 pat_first_; // exposition only
    RandomAccessIterator1 pat_last_;  // exposition only
    Hash hash_;                       // exposition only
    BinaryPredicate pred_;           // exposition only
};
```

boyer_moore_searcher(RandomAccessIterator1 pat_first,
    RandomAccessIterator1 pat_last,
    Hash hf = Hash(),
    BinaryPredicate pred = BinaryPredicate());

Requires: The value type of RandomAccessIterator1 shall satisfy the Cpp17DefaultConstructible requirements, the Cpp17CopyConstructible requirements, and the Cpp17CopyAssignable requirements.

Requires: For any two values A and B of the type iterator_traits<RandomAccessIterator1>::value_type, if pred(A, B) == true, then hf(A) == hf(B) shall be true.

Effects: Constructs a boyer_moore_searcher object, initializing 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;

Requires: RandomAccessIterator1 and RandomAccessIterator2 shall 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
--- i is the first iterator in the range \([\text{first}, \text{last} - (\text{pat_last}_- - \text{pat_first}_-))\) such that for every non-negative integer \(n\) less than \(\text{pat_last}_- - \text{pat_first}_-\) the following condition holds:

\[\text{pred}(*(\text{i} + n), *(\text{pat_first}_- + n)) \neq \text{false},\]

--- \(j == \text{next}(i, \text{distance}(\text{pat_first}_-, \text{pat_last}_-))\).

Returns \(\text{make_pair}((\text{first}, \text{first})\) if \([\text{pat_first}_-, \text{pat_last}_-\) is empty, otherwise returns \(\text{make_pair}((\text{last}, \text{last})\) if no such iterator is found.

\[\text{Complexity: At most } (\text{last} - \text{first}) \ast (\text{pat_last}_- - \text{pat_first}_-) \text{ applications of the predicate.}\]

### 20.14.17.3 Class template boyer_moore_horspool_searcher

```cpp
template<class RandomAccessIterator1, 
class Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>, 
class BinaryPredicate = equal_to<>>
class boyer_moore_horspool_searcher {
public:
    boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first,
                                RandomAccessIterator1 pat_last, 
                                Hash hf = Hash(), 
                                BinaryPredicate pred = BinaryPredicate());

    template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
    operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

private:
    RandomAccessIterator1 pat_first_;   // exposition only
    RandomAccessIterator1 pat_last_;    // exposition only
    Hash hash_;                        // exposition only
    BinaryPredicate pred_;             // exposition only
};
```

--- Requires: The value type of \(\text{RandomAccessIterator1}\) shall satisfy the Cpp17DefaultConstructible, Cpp17CopyConstructible, and Cpp17CopyAssignable requirements.

--- Requires: For any two values \(A\) and \(B\) of the type iterator_traits<RandomAccessIterator1>::value_type, if \(\text{pred}(A, B) == \text{true}\), then \(\text{hf}(A) == \text{hf}(B)\) shall be true.

--- Effects: Constructs a \(\text{boyer_moore_horspool_searcher}\) object, initializing \(\text{pat_first}_-\) with \(\text{pat_first}\), \(\text{pat_last}_-\) with \(\text{pat_last}\), \(\text{hash}_-\) with \(\text{hf}\), and \(\text{pred}_-\) with \(\text{pred}\).

--- Throws: Any exception thrown by the copy constructor of \(\text{RandomAccessIterator1}\), or by the default constructor, copy constructor, or the copy assignment operator of the value type of \(\text{RandomAccessIterator1}\) or the copy constructor or \(\text{operator()}\) of \(\text{BinaryPredicate}\) or \(\text{Hash}\). May throw \text{bad_alloc} if additional memory needed for internal data structures cannot be allocated.

```cpp
template<class RandomAccessIterator2>
pair<RandomAccessIterator2, RandomAccessIterator2>
operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
```

--- Requires: \(\text{RandomAccessIterator1}\) and \(\text{RandomAccessIterator2}\) shall 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

--- \(i \) is the first iterator \(i\) in the range \([\text{first}, \text{last} - (\text{pat_last}_- - \text{pat_first}_-))\) such that for every non-negative integer \(n\) less than \(\text{pat_last}_- - \text{pat_first}_-\) the following condition holds:

\[\text{pred}(*(\text{i} + n), *(\text{pat_first}_- + n)) \neq \text{false},\]

--- \(j == \text{next}(i, \text{distance}(\text{pat_first}_-, \text{pat_last}_-))\).
Returns \texttt{make_pair(first, first)} if \texttt{[pat\_first\_, pat\_last\_]} is empty, otherwise returns \texttt{make\_pair(last, last)} if no such iterator is found.

\textit{Complexity:} At most \((last - first) \times (pat\_last\_ - pat\_first\_)\) applications of the predicate.

\textbf{20.14.18 Class template hash}\quad [\texttt{unord.hash}]

\textit{1} The unordered associative containers defined in 22.5 use specializations of the class template \texttt{hash (20.14.1)} as the default hash function.

\textit{2} Each specialization of \texttt{hash} is either enabled or disabled, as described below. \textit{[Note: Enabled specializations meet the Cpp17Hash requirements, and disabled specializations do not. \textemdash\ end note]} Each header that declares the template \texttt{hash} provides enabled specializations of \texttt{hash} for \texttt{nullptr_t} and all cv-unqualified arithmetic, enumeration, and pointer types. For any type \texttt{Key} for which neither the library nor the user provides an explicit or partial specialization of the class template \texttt{hash, hash<Key>} is disabled.

\textit{3} If the library provides an explicit or partial specialization of \texttt{hash<Key>}, that specialization is enabled except as noted otherwise, and its member functions are \texttt{noexcept} except as noted otherwise.

\textit{4} If \texttt{H} is a disabled specialization of \texttt{hash}, these values are \texttt{false: is\_default\_constructible\_v<H>, is\_move\_constructible\_v<H>, is\_copy\_assignable\_v<H>}, and \texttt{is\_move\_assignable\_v<H>}. Disabled specializations of \texttt{hash} are not function object types (20.14). \textit{[Note: This means that the specialization of \texttt{hash} exists, but any attempts to use it as a Cpp17Hash will be ill-formed. \textemdash\ end note]}

\textit{5} An enabled specialization \texttt{hash<Key>} will:

\begin{enumerate}[\textit{(5.1)}]
\item satisfy the \texttt{Cpp17Hash} requirements (Table 32), with \texttt{Key} as the function call argument type, the \texttt{Cpp17DefaultConstructible} requirements (Table 25), the \texttt{Cpp17CopyAssignble} requirements (Table 29),
\item be swappable (16.5.3.2) for lvalues,
\item satisfy the requirement that if \texttt{k1 == k2} is \texttt{true}, \texttt{h(k1) == h(k2)} is also \texttt{true}, where \texttt{h} is an object of type \texttt{hash<Key>} and \texttt{k1} and \texttt{k2} are objects of type \texttt{Key};
\item satisfy the requirement that the expression \texttt{h(k)}, where \texttt{h} is an object of type \texttt{hash<Key>} and \texttt{k} is an object of type \texttt{Key}, shall not throw an exception unless \texttt{hash<Key>} is a program-defined specialization that depends on at least one program-defined type.
\end{enumerate}

\textbf{20.15 Metaprogramming and type traits}\quad [\texttt{meta}]

\textit{1} 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.

\textit{2} All functions specified in this subclause are signal-safe (17.13.4).

\textbf{20.15.1 Requirements}\quad [\texttt{meta.rqmts}]

\textit{1} A \texttt{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 \texttt{Cpp17DefaultConstructible, Cpp17CopyConstructible}, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template \texttt{integral\_constant (20.15.3)}, with the arguments to the template \texttt{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 \texttt{Cpp17UnaryTypeTrait}.

\textit{2} A \texttt{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 \texttt{Cpp17DefaultConstructible, Cpp17CopyConstructible}, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template \texttt{integral\_constant (20.15.3)}, with the arguments to the template \texttt{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 \texttt{Cpp17BinaryTypeTrait}.
A `Cpp17TransformationTrait` modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a publicly accessible nested type named `type`, which shall be a synonym for the modified type.

Unless otherwise specified, the behavior of a program that adds specializations for any of the templates specified in this subclause 20.15 is undefined.

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:

1. An instantiation of a template specified in this subclause directly or indirectly depends on an incompletely-defined object type `T`, and
2. That instantiation could yield a different result were `T` hypothetically completed.

### 20.15.2 Header `<type_traits>` synopsis

```cpp
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_volatile;
    template<class T> struct is_trivial;
    template<class T> struct is_trivially_copyable;
    template<class T> struct is_standard_layout;
    template<class T> struct is_empty;
    template<class T> struct is_polymorphic;
    template<class T> struct is_abstract;
    template<class T> struct is_final;
    template<class T> struct is_aggregate;
    template<class T> struct is_signed;
    template<class T> struct is_unsigned;
    template<class T> struct is_bounded_array;
}```
template<class T> struct is_unbounded_array;

template<class T, class... Args> struct is_constructible;
template<class T> struct is_default_constructible;
template<class T> struct is_copy_constructible;
template<class T> struct is_move_constructible;

template<class T, class U> struct is_assignable;
template<class T> struct is_copy_assignable;
template<class T> struct is_move_assignable;

// 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 Fn, class... ArgTypes> struct is_invocable;
template<class R, class Fn, class... ArgTypes> struct is_invocable_r;

template<class Fn, class... ArgTypes> struct is_nothrow_invocable;
template<class R, class Fn, class... ArgTypes> struct is_nothrow_invocable_r;

// 20.15.7.1, 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;

// 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;

// 20.15.7.3, sign modifications
template<class T> struct make_signed;
template<class T> struct make_unsigned;

// 20.15.7.4, array modifications
template<class T> struct remove_extent;
template<class T> struct remove_all_extents;

// 20.15.7.5, pointer modifications
template<class T> struct remove_pointer;
template<class T> struct add_pointer;

// 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 decay;
template<\texttt{bool}, \texttt{class T} = \texttt{void}> \texttt{struct enable\_if;}
template<\texttt{bool}, \texttt{class T}, \texttt{class F} > \texttt{struct conditional;}
template<\texttt{class... T} > \texttt{struct common\_type;}
template<\texttt{class T}, \texttt{class U}, \texttt{template<class> class T\texttt{Qual}}, \texttt{template<class> class U\texttt{Qual}}>
\texttt{struct basic\_common\_reference \{ \};}
template<\texttt{class... T} > \texttt{struct common\_reference;}
template<\texttt{class T} > \texttt{struct underlying\_type;}
template<\texttt{class Fn, class... ArgTypes} > \texttt{struct invoke\_result;}

template<\texttt{class T}>
\texttt{using type\_identity\_t = typename type\_identity\lt T\rt::\texttt{type;}}
template<\texttt{size\_t Len, size\_t Align = default\_alignment} // see \texttt{20.15.7.6}
\texttt{using aligned\_storage\_t = typename aligned\_storage\lt Len, Align\rt::\texttt{type;}}
template<\texttt{size\_t Len}, \texttt{class... Types}>
\texttt{using aligned\_union\_t = typename aligned\_union\lt Len, Types...\rt::\texttt{type;}}
template<\texttt{class T}>
\texttt{using remove\_cvref\_t = typename remove\_cvref\lt T\rt::\texttt{type;}}
template<\texttt{class T}>
\texttt{using decay\_t = typename decay\lt T\rt::\texttt{type;}}
template<\texttt{bool b, class T = \texttt{void}}>
\texttt{using enable\_if\_t = typename enable\_if\lt b, T\rt::\texttt{type;}}
template<\texttt{bool b, class T, class F}>
\texttt{using conditional\_t = typename conditional\lt b, T, F\rt::\texttt{type;}}
template<\texttt{class... T}>
\texttt{using common\_type\_t = typename common\_type\lt T...\rt::\texttt{type;}}
template<\texttt{class... T}>
\texttt{using common\_reference\_t = typename common\_reference\lt T...\rt::\texttt{type;}}
template<\texttt{class T}>
\texttt{using underlying\_type\_t = typename underlying\_type\lt T\rt::\texttt{type;}}
template<\texttt{class Fn, class... ArgTypes}>
\texttt{using invoke\_result\_t = typename invoke\_result\lt Fn, ArgTypes...\rt::\texttt{type;}}
template<\texttt{class...}>
\texttt{using void\_t = void;}

\texttt{// 20.15.8, logical operator traits}
template<\texttt{class... B} > \texttt{struct conjunction;}
template<\texttt{class... B} > \texttt{struct disjunction;}
template<\texttt{class B} > \texttt{struct negation;}

\texttt{// 20.15.9, endian}
\texttt{enum class endian \{}
\texttt{  little = see below,}
\texttt{  big = see below,}
\texttt{  native = see below }
\texttt{\};}

\texttt{// 20.15.4.1, primary type categories}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_void\_v = is\_void\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_null\_pointer\_v = is\_null\_pointer\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_integral\_v = is\_integral\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_floating\_point\_v = is\_floating\_point\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_array\_v = is\_array\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_pointer\_v = is\_pointer\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_lvalue\_reference\_v = is\_lvalue\_reference\lt T\rt::\texttt{value;}}
template<\texttt{class T}>
\texttt{inline constexpr bool is\_rvalue\_reference\_v = is\_rvalue\_reference\lt T\rt::\texttt{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_compound_v = is_compound<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>
  inline constexpr bool is_move_constructible_v = is_move_constructible<T>::value;

template<class T, class U>
  inline constexpr bool is_assignable_v = is_assignable<T, U>::value;

template<class T>
  inline constexpr bool is_copy_assignable_v = is_copy_assignable<T>::value;

template<class T>
  inline constexpr bool is_move_assignable_v = is_move_assignable<T>::value;

template<class T, class U>
  inline constexpr bool is_swappable_with_v = is_swappable_with<T, U>::value;

template<class T>
  inline constexpr bool is_swappable_v = is_swappable<T>::value;

template<class T>
  inline constexpr bool is_destructible_v = is_destructible<T>::value;

template<class T, class... Args>
  inline constexpr bool is_trivially_constructible_v
  = is_trivially_constructible<T, Args...>::value;

template<class T>
  inline constexpr bool is_trivially_default_constructible_v
  = is_trivially_default_constructible<T>::value;

template<class T>
  inline constexpr bool is_trivially_copy_constructible_v
  = is_trivially_copy_constructible<T>::value;

template<class T>
  inline constexpr bool is_trivially_move_constructible_v
  = is_trivially_move_constructible<T>::value;

template<class T, class U>
  inline constexpr bool is_trivially_assignable_v = is_trivially_assignable<T, U>::value;

template<class T>
  inline constexpr bool is_trivially_copy_assignable_v
  = is_trivially_copy_assignable<T>::value;

template<class T>
  inline constexpr bool is_trivially_move_assignable_v
  = is_trivially_move_assignable<T>::value;

template<class T>
  inline constexpr bool is_trivially_destructible_v = is_trivially_destructible<T>::value;

template<class T, class... Args>
  inline constexpr bool is_nothrow_constructible_v
  = is_nothrow_constructible<T, Args...>::value;

template<class T>
  inline constexpr bool is_nothrow_default_constructible_v
  = is_nothrow_default_constructible<T>::value;

template<class T>
  inline constexpr bool is_nothrow_copy_constructible_v
  = is_nothrow_copy_constructible<T>::value;

template<class T>
  inline constexpr bool is_nothrow_move_constructible_v
  = is_nothrow_move_constructible<T>::value;

template<class T, class U>
  inline constexpr bool is_nothrow_assignable_v = is_nothrow_assignable<T, U>::value;

template<class T>
  inline constexpr bool is_nothrow_copy_assignable_v
  = is_nothrow_copy_assignable<T>::value;

template<class T>
  inline constexpr bool is_nothrow_move_assignable_v
  = is_nothrow_move_assignable<T>::value;

template<class T, class U>
  inline constexpr bool is_nothrow_swappable_with_v = is_nothrow_swappable_with<T, U>::value;

template<class T>
  inline constexpr bool is_nothrow_swappable_v = is_nothrow_swappable<T>::value;

template<class T>
  inline constexpr bool is_nothrow_destructible_v = is_nothrow_destructible<T>::value;

template<class T>
  inline constexpr bool has_virtualDestructor_v = has_virtualDestructor<T>::value;
template<class T>
inline constexpr bool has_unique_object_representations_v
   = has_unique_object_representations<T>::value;

// 20.15.5, type property queries
template<class T>
inline constexpr size_t alignment_of_v = alignment_of<T>::value;
template<class T>
inline constexpr size_t rank_v = rank<T>::value;
template<class T, unsigned I = 0>
inline constexpr size_t extent_v = extent<T, I>::value;

// 20.15.6, type relations
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 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 Fn, class... ArgTypes>
inline constexpr bool is_nothrow_invocable_v = is_nothrow_invocable<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;

// 20.15.8, logical operator traits
template<class... B>
inline constexpr bool conjunction_v = conjunction<B...>::value;
template<class... B>
inline constexpr bool disjunction_v = disjunction<B...>::value;
template<class B>
inline constexpr bool negation_v = negation<B>::value;

// 20.15.10, constant evaluation context
constexpr bool is_constant_evaluated() noexcept;
}

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; }
   };
}

1 The class template integral_constant, alias template bool_constant, and its associated typedef-names true_type and false_type are used as base classes to define the interface for various type traits.

20.15.4 Unary type traits [meta.unary]
1 This subclause contains templates that may be used to query the properties of a type at compile time.
2 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

1 The primary type categories correspond to the descriptions given in subclause 6.7 of the C++ standard.

2 For any given type \( T \), the result of applying one of these templates to \( T \) and to \( cv \ T \) shall yield the same result.

3 [Note: For any given type \( T \), exactly one of the primary type categories has a value member that evaluates to true. — end note]

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T&gt; struct is_void;</code></td>
<td>( T ) is void</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_null_pointer;</code></td>
<td>( T ) is <code>nullptr_t</code> (6.7.1)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_integral;</code></td>
<td>( T ) is an integral type (6.7.1)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_floating_point;</code></td>
<td>( T ) is a floating-point type (6.7.1)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_array;</code></td>
<td>( T ) is an array type (6.7.2) of known or unknown extent</td>
<td>Class template ( array ) (22.3.7) is not an array type.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_pointer;</code></td>
<td>( T ) is a pointer type (6.7.2)</td>
<td>Includes pointers to functions but not pointers to non-static members.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_lvalue_reference;</code></td>
<td>( T ) is an lvalue reference type (9.2.3.2)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_rvalue_reference;</code></td>
<td>( T ) is an rvalue reference type (9.2.3.2)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_member_object_pointer;</code></td>
<td>( T ) is a pointer to data member</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_member_function_pointer;</code></td>
<td>( T ) is a pointer to member function</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_enum;</code></td>
<td>( T ) is an enumeration type (6.7.2)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_union;</code></td>
<td>( T ) is a union type (6.7.2)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_class;</code></td>
<td>( T ) is a non-union class type (6.7.2)</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_function;</code></td>
<td>( T ) is a function type (6.7.2)</td>
<td></td>
</tr>
</tbody>
</table>

20.15.4.2 Composite type traits

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in subclause 6.7.

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>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T&gt; struct is_reference;</code></td>
<td>( T ) is an lvalue reference or an rvalue reference</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct is_arithmetic;</code></td>
<td>( T ) is an arithmetic type (6.7.1)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 46 — Composite type category predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_fundamental;</td>
<td>T is a fundamental type (6.7.1)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_object;</td>
<td>T is an object type (6.7)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_scalar;</td>
<td>T is a scalar type (6.7)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_compound;</td>
<td>T is a compound type (6.7.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_pointer;</td>
<td>T is a pointer-to-member type (6.7.2)</td>
<td></td>
</tr>
</tbody>
</table>

### 20.15.4.3 Type properties

1. These templates provide access to some of the more important properties of types.
2. It is unspecified whether the library defines any full or partial specializations of any of these templates.
3. For all of the class templates X declared in this subclause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of X require that the argument is a complete type.
4. For the purpose of defining the templates in this subclause, a function call expression `declval<T>()` for any type T is considered to be a trivial (6.7, 11.3.3) function call that is not an odr-use (6.2) of `declval` in the context of the corresponding definition notwithstanding the restrictions of 20.2.6.

### Table 47 — Type property predicates

<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 (6.7.3)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_volatile;</td>
<td>T is volatile-qualified (6.7.3)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivial;</td>
<td>T is a trivial type (6.7)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or <code>cv</code> void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_copyable;</td>
<td>T is a trivially copyable type (6.7)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or <code>cv</code> void.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_standard_layout;</td>
<td>T is a standard-layout type (6.7)</td>
<td><code>remove_all_extents_t&lt;T&gt;</code> shall be a complete type or <code>cv</code> 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 <code>is_empty_v&lt;B&gt;</code> is false.</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_polymorphic;</td>
<td>T is a polymorphic class (11.6.2)</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 (11.6.3)</td>
<td>If T is a non-union class type, T shall be a complete type.</td>
</tr>
</tbody>
</table>
### Table 47 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{template&lt;class T&gt; struct is_final;} )</td>
<td>( T ) is a class type marked with the <code>class-virt-specifier final</code> (Clause 11).</td>
<td>If ( T ) is a class type, ( T ) shall be a complete type.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_aggregate;} )</td>
<td>( T ) is an aggregate type (9.3.1)</td>
<td>( \text{remove_all_extents_t&lt;}T\text{&gt;} ) shall be a complete type or ( \text{cv void} ).</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_signed;} )</td>
<td>If ( \text{is_arithmetic_v&lt;T&gt;} ) is true, the same result as ( T(-1) &lt; T(0) ); otherwise, ( \text{false} )</td>
<td>( T ) shall be a complete type, ( \text{cv void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_unsigned;} )</td>
<td>If ( \text{is_arithmetic_v&lt;T&gt;} ) is true, the same result as ( T(0) &lt; T(-1) ); otherwise, ( \text{false} )</td>
<td>( T ) shall be a complete type, ( \text{cv void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_bounded_array;} )</td>
<td>( T ) is an array type of known bound (9.2.3.4)</td>
<td>( T ) and all types in the template parameter pack ( \text{Args...} ) shall be complete types, ( \text{cv void} ), or arrays of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_unbounded_array;} )</td>
<td>( T ) is an array type of unknown bound (9.2.3.4)</td>
<td>( T ) shall be a complete type, ( \text{cv void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T, class... Args&gt; struct is_constructible;} )</td>
<td>For a function type ( T ) or for a ( \text{cv void} ) type ( T ), ( \text{is_constructible_v&lt;T, Args...&gt;} ) is false, otherwise see below</td>
<td>( T ) and all types in the template parameter pack ( \text{Args...} ) shall be complete types, ( \text{cv void} ), or arrays of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_default_constructible;} )</td>
<td>( \text{is_constructible_v&lt;T&gt;} ) is true.</td>
<td>( T ) shall be a complete type, ( \text{cv void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_copy_constructible;} )</td>
<td>For a referenceable type ( T ) (16.3.22), the same result as ( \text{is_constructible_v&lt;T, const T&amp;&gt;} ), otherwise ( \text{false} )</td>
<td>( T ) shall be a complete type, ( \text{cv void} ), or an array of unknown bound.</td>
</tr>
<tr>
<td>( \text{template&lt;class T&gt; struct is_move_constructible;} )</td>
<td>For a referenceable type ( T ), the same result as ( \text{is_constructible_v&lt;T, T&amp;&amp;&gt;} ), otherwise ( \text{false} )</td>
<td>( T ) shall be a complete type, ( \text{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;</td>
<td>The expression <code>declval&lt;T&gt;() = declval&lt;U&gt;()</code> 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></td>
</tr>
<tr>
<td>struct is_assignable;</td>
<td></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;</td>
<td></td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>struct is_copyAssignable;</td>
<td>For a referenceable type T, the same result as <code>is_assignable_v&lt;T&amp;, const T&amp;&gt;, otherwise false.</code></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td></td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>struct is_moveAssignable;</td>
<td>For a referenceable type T, the same result as <code>is_assignable_v&lt;T&amp;, T&amp;&amp;&gt;, otherwise false.</code></td>
<td></td>
</tr>
</tbody>
</table>
### Table 47 — 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, class U&gt;</code></td>
<td>The expressions <code>swap(declval&lt;T&gt;()</code>, <code>declval&lt;U&gt;()</code>) and <code>swap(declval&lt;T&gt;(), declval&lt;U&gt;())</code> 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 <code>T</code> and <code>U</code>. Only the validity of the immediate context of the <code>swap</code> 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 &quot;immediate context&quot; and can result in the program being ill-formed. — end note]</td>
<td></td>
</tr>
<tr>
<td><code>struct is_swappable_with;</code></td>
<td><code>T</code> and <code>U</code> shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td>For a referenceable type <code>T</code>, the same result as <code>is_swappable_with_v&lt;T&amp;, T&amp;&gt;</code>, otherwise <code>false</code>.</td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>struct is_swappable;</code></td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T&gt;</code></td>
<td>Either <code>T</code> is a reference type, or <code>T</code> is a complete object type for which the expression <code>declval&lt;U&amp;&gt;()</code>, <code>~U()</code> is well-formed when treated as an unevaluated operand (7.2), where <code>U</code> is <code>remove_all_extents_t&lt;T&gt;</code>.</td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td><code>struct is_destructible;</code></td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
<td></td>
</tr>
<tr>
<td><code>template&lt;class T, class... Args&gt;</code></td>
<td><code>is_constructible_v&lt;T, Args...&gt; is true and the variable definition for </code>is_constructible<code>, as defined below, is known to call no operation that is not trivial (6.7, 11.3.3).</code></td>
<td><code>T</code> and all types in the template parameter pack <code>Args</code> shall be complete types, <code>cv void</code>, 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 (6.7, 11.3.3).</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 (7.6.2.8).</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 47 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_move_constructible;</td>
<td>For a referenceable type T, the same result as is_nothrow_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_nothrow_assignable;</td>
<td>is_assignable_v&lt;T, U&gt; is true and the assignment is known not to throw any exceptions (7.6.2.8).</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_copy_assignable;</td>
<td>For a referenceable type T, the same result as is_nothrow_assignable_v&lt;T&amp;, const T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_move_assignable;</td>
<td>For a referenceable type T, the same result as is_nothrow_assignable_v&lt;T&amp;, T&amp;&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_nothrow_swappable_with;</td>
<td>is_swappable_with_v&lt;T, U&gt; is true and each swap expression of the definition of is_swappable_with&lt;T, U&gt; is known not to throw any exceptions (7.6.2.8).</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_swappable;</td>
<td>For a referenceable type T, the same result as is_nothrow_swappable_v&lt;T, T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_destructible;</td>
<td>is_destructible_v&lt;T&gt; is true and the indicated destructor is known not to throw any exceptions (7.6.2.8).</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct has_virtual_destructor;</td>
<td>T has a virtual destructor (11.3.6)</td>
<td>If T is a non-union class type, T shall be a complete type.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct has_unique_object_representations;</td>
<td>For an array type T, the same result as has_unique_object_representations_v&lt;remove_all_extents_v&lt;T&gt;&gt;, otherwise see below.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
</tbody>
</table>

[Example:]

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_const_v&lt;const volatile int&gt;</td>
<td>// true</td>
</tr>
<tr>
<td>is_const_v&lt;const int&gt;</td>
<td>// false</td>
</tr>
<tr>
<td>is_const_v&lt;const int&gt;&amp;</td>
<td>// false</td>
</tr>
<tr>
<td>is_const_v&lt;int[3]&gt;</td>
<td>// false</td>
</tr>
<tr>
<td>is_const_v&lt; const int[3]&gt;</td>
<td>// true</td>
</tr>
</tbody>
</table>

— end example]

[Example:]
remove_const_t<const volatile int>  // volatile int
remove_const_t<const int* const>  // const int*
remove_const_t<const int&>  // const int&

— end example]

Example:

// Given:
struct P final { };  
union U1 { };  
union U2 final { };

// the following assertions hold:
static_assert(!is_final_v<int>);  
static_assert(is_final_v<P>);  
static_assert(!is_final_v<U1>);  
static_assert(is_final_v<U2>);

— end example]

The predicate condition for a template specialization is_constructible<T, Args...> shall be satisfied if and only if the following variable definition would be well-formed for some invented variable t:

T t(declval<Args>()...);

[Note: 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]

The predicate condition for a template specialization has_unique_object_representations<T> shall be satisfied if and only if:

— T is trivially copyable, and
— 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

This subclause contains templates that may be used to query properties of types at compile time.

Table 48 — Type property queries

<table>
<thead>
<tr>
<th>Template</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>alignof(T).</td>
</tr>
<tr>
<td>struct alignment_of;</td>
<td>Requires: alignof(T) shall be a valid expression (7.6.2.7)</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>If T names an array type, an integer value representing the number of</td>
</tr>
<tr>
<td>struct rank;</td>
<td>dimensions of T; otherwise, 0.</td>
</tr>
<tr>
<td>template&lt;class T,</td>
<td>If T is not an array type, or if it has rank less than or equal to I,</td>
</tr>
<tr>
<td>unsigned I = 0&gt;</td>
<td>or if I is 0 and T has type “array of unknown bound of U”, then 0;</td>
</tr>
<tr>
<td>struct extent;</td>
<td>otherwise, the bound (9.2.3.4) of the I(^{th}) dimension of T,</td>
</tr>
<tr>
<td></td>
<td>where indexing of I is zero-based</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:
### 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>
<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 (11.6) 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, arrays of unknown bound, or <code>cv void</code> types.</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 (7.6.2.8)</td>
<td>From and To shall be complete types, arrays of unknown bound, or <code>cv void</code> types.</td>
</tr>
<tr>
<td>template&lt;class Fn, class... ArgTypes&gt; struct is_invocable;</td>
<td>The expression <code>INVOKEDeclval&lt;Fn&gt;()</code>, <code>Declval&lt;ArgTypes&gt;()</code>... is well-formed when treated as an unevaluated operand</td>
<td>Fn and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class R, class Fn, class... ArgTypes&gt; struct is_invocable_r;</td>
<td>The expression <code>INVOKER&lt;(Declval&lt;Fn&gt;(), Declval&lt;ArgTypes&gt;()...)</code> is well-formed when treated as an unevaluated operand</td>
<td>Fn, R, and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>
Table 49 — Type relationship predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class Fn, class... ArgTypes&gt; struct is_nothrow_invocable;</td>
<td>is_invocable_v&lt;Fn, ArgTypes...&gt; is true and the expression INVOKE(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is known not to throw any exceptions (7.6.2.8)</td>
<td>Fn and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class R, class Fn, class... ArgTypes&gt; struct is_nothrow_invocable_r;</td>
<td>is_invocable_r_v&lt;R, Fn, ArgTypes...&gt; is true and the expression INVOKE&lt;R&gt;(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is known not to throw any exceptions (7.6.2.8)</td>
<td>Fn, R, and all types in the template parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>

3 For the purpose of defining the templates in this subclause, a function call expression declval<T>() for any type T is considered to be a trivial (6.7, 11.3.3) function call that is not an odr-use (6.2) of declval in the context of the corresponding definition notwithstanding the restrictions of 20.2.6.

4 [Example:

```cpp
struct B {};
struct B1 : B {};
struct B2 : B {};
struct D : private B1, private B2 {};

is_base_of_v<B, D>    // true
is_base_of_v<const B, D> // true
is_base_of_v<B, const D> // true
is_base_of_v<B, B>    // false
is_base_of_v<B&, D&>  // false
is_base_of_v<B[3], D[3]> // false
is_base_of_v<int, int> // false

@end example]

5 The predicate condition for a template specialization 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 To and From. Only the validity of the immediate context of the expression of the return statement (8.6.3) (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]

1 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 Cpp17TransformationTrait (20.15.1).

20.15.7.1 Const-volatile modifications [meta.trans.cv]
Table 50 — Const-volatile modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct remove_const;</td>
<td>The member typedef type names the same type as T except that any top-level const-qualifier has been removed. [Example: remove_const_t&lt;const volatile int&gt; evaluates to volatile int, whereas remove_const_t&lt;const int*&gt; evaluates to const int*. — end example]</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct remove_volatile;</td>
<td>The member typedef type names the same type as T except that any top-level volatile-qualifier has been removed. [Example: remove_volatile_t&lt;const volatile int&gt; evaluates to const int, whereas remove_volatile_t&lt;volatile int*&gt; evaluates to volatile int*. — end example]</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct remove_cv;</td>
<td>The member typedef type shall be the same as T except that any top-level cv-qualifier has been removed. [Example: remove_cv_t&lt;const volatile int&gt; evaluates to int, whereas remove_cv_t&lt;const volatile int*&gt; evaluates to volatile int*. — end example]</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct add_const;</td>
<td>If T is a reference, function, or top-level const-qualified type, then type names the same type as T, otherwise T const.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct add_volatile;</td>
<td>If T is a reference, function, or top-level volatile-qualified type, then type names the same type as T, otherwise T volatile.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct add_cv;</td>
<td>The member typedef type names the same type as add_const_t&lt;add_volatile_t&lt;T&gt;&gt;.</td>
</tr>
</tbody>
</table>

20.15.7.2 Reference modifications

Table 51 — Reference modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct remove_reference;</td>
<td>If T has type “reference to T1” then the member typedef type names T1; otherwise, type names T.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct add_lvalue_reference;</td>
<td>If T names a referenceable type (16.3.22) then the member typedef type names T&amp;; otherwise, type names T. [Note: This rule reflects the semantics of reference collapsing (9.2.3.2). — end note]</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct add_rvalue_reference;</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 (9.2.3.2). For example, when a type T names a type T1&amp;, 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 52 — Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct make_signed;</td>
<td>If T names a (possibly cv-qualified) signed integer type (6.7.1) 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 (6.7.4) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. Requires: T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type.</td>
</tr>
</tbody>
</table>
Table 52 — Sign modifications (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template<class T>
| struct make_unsigned; | If T names a (possibly cv-qualified) unsigned integer type (6.7.1) 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 unsign integer type, with the same cv-qualifiers as T; otherwise, type names the unsign integer type with smallest rank (6.7.4) for which sizeof(T) == sizeof(type), with the same cv-qualifiers as T. Requires: T shall be a (possibly cv-qualified) integral type or enumeration but not a bool type. |

20.15.7.4 Array modifications

Table 53 — Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template<class T>
| struct remove_extent; | If T names a type “array of U”, the member typedef 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] |
| template<class T>
| struct remove_all_extents; | If T is “multi-dimensional array of U”, the resulting member typedef type is U, otherwise T. |

1 Example:

// the following assertions hold:
assert((is_same_v<remove_extent_t<int>, int>>));
assert((is_same_v<remove_extent_t<int[2], int>>));
assert((is_same_v<remove_extent_t<int[2][3], int[3]>>));
assert((is_same_v<remove_extent_t<int[][], int[]>>));
—end example]

2 Example:

// the following assertions hold:
assert((is_same_v<remove_all_extents_t<int>, int>>));
assert((is_same_v<remove_all_extents_t<int[2], int>>));
assert((is_same_v<remove_all_extents_t<int[2][3], int>>));
assert((is_same_v<remove_all_extents_t<int[][], int>>));
—end example]

20.15.7.5 Pointer modifications

Table 54 — Pointer modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| template<class T>
| struct remove_pointer; | If T has type “(possibly cv-qualified) pointer to T1” then the member typedef type names T1; otherwise, it names T. |
| template<class T>
| struct add_pointer; | If T names a referenceable type (16.3.22) or a cv void type then the member typedef type names the same type as remove_reference_t<T>*; otherwise, type names T. |

20.15.7.6 Other transformations

§ 20.15.7.6
Table 55 — 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>
<tr>
<td>template&lt;size_t Len, size_t Align = default-alignment&gt; struct aligned_storage;</td>
<td>The value of default-alignment shall be the most stringent alignment requirement for any C++ object type whose size is no greater than Len (6.7). 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. Requires: Len shall not be zero. Align shall be equal to alignof(T) for some type T or to default-alignment.</td>
</tr>
<tr>
<td>template&lt;size_t Len, class... Types&gt; struct aligned_union;</td>
<td>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. Requires: At least one type is provided. Each type in the template parameter pack Types shall be a complete object type.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct remove_cvref;</td>
<td>The member typedef type names the same type as remove_cv_t&lt;remove_reference_t&lt;T&gt;&gt;.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct decay;</td>
<td>Let U be remove_reference_t&lt;T&gt;. If is_array_v&lt;U&gt; is true, the member typedef type shall equal remove_extent_t&lt;U&gt;*. If is_function_v&lt;U&gt; is true, the member typedef type shall equal add_pointer_t&lt;U&gt;. Otherwise the member typedef type equals remove_cv_t&lt;U&gt;. [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 lvalue is used as an rvalue, but also strips cv-qualifiers from class types in order to more closely model by-value argument passing. —end note]</td>
</tr>
<tr>
<td>template&lt;bool B, class T = void&gt; struct enable_if;</td>
<td>If B is true, the member typedef type shall equal T; otherwise, there shall be no member type.</td>
</tr>
<tr>
<td>template&lt;bool B, class T, class F&gt; struct conditional;</td>
<td>If B is true, the member typedef type shall equal T. If B is false, the member typedef type shall equal F.</td>
</tr>
<tr>
<td>template&lt;class... T&gt; struct common_type;</td>
<td>Unless this trait is specialized (as specified in Note B, below), the member typedef type shall be 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.</td>
</tr>
<tr>
<td>template&lt;class, class, template&lt;class&gt; class, template&lt;class&gt; class&gt; struct basic_common_reference;</td>
<td>Unless this trait is specialized (as specified in Note D, below), there shall be no member type.</td>
</tr>
<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 (9.6); otherwise, there is no member type. Mandates: T is not an incomplete enumeration type.</td>
</tr>
</tbody>
</table>

§ 20.15.7.6
Table 55 — Other transformations (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class Fn, class... ArgTypes&gt; struct invoke_result;</td>
<td>If the expression <code>INVOKE</code>(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...) is well-formed when treated as an unevaluated operand (7.2), the member typedef <code>type</code> names the type <code>decltype(INVOKE</code>(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...)); otherwise, there shall be no member <code>type</code>. Access checking is performed as if in a context unrelated to <code>Fn</code> and <code>ArgTypes</code>. 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>
</tbody>
</table>

Requires: `Fn` and all types in the template parameter pack `ArgTypes` shall be complete types, `cv void`, or arrays of unknown bound.

---

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 Let:

1. `CREF(A)` be `add_lvalue_reference_t<const remove_reference_t<A>>`,
2. `XREF(A)` denote a unary alias template `T` such that `T<U>` denotes the same type as `U` with the addition of `A`’s cv and reference qualifiers, for a non-reference `cv-unqualified` type `U`,
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]
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:

1. 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.
2. Otherwise, let `C` be `remove_reference_t<COMMON_REF(X, Y)>&`. If `A` and `B` are both rvalue reference types, `C` is well-formed, and `is_convertible_v<y<A, C>`, `&yS` is `is_convertible_v<x<B, C>`, `true`, then `COMMON_REF(A, B)` is `C`.
3. Otherwise, let `D` be `COMMON_REF(const X, Y)`. If `A` is an rvalue reference and `B` is an lvalue reference and `D` is well-formed and `is_convertible_v<x<A, D>`, `true`, then `COMMON_REF(A, B)` is `D`.
4. Otherwise, if `A` is an lvalue reference and `B` is an rvalue reference, then `COMMON_REF(A, B)` is `COMMON_REF(B, A)`.
5. Otherwise, `COMMON_REF(A, B)` is ill-formed.

If any of the types computed above is ill-formed, then `COMMON_REF(A, B)` is ill-formed.

3 Note A: For the `common_type` trait applied to a template parameter pack `T` of types, the member `type` shall be either defined or not present as follows:

1. If `sizeof...(T)` is zero, there shall be no member `type`.
2. If `sizeof...(T)` is one, let `T0` denote the sole type constituting the pack `T`. The member `typedef-name type` shall denote the same type, if any, as `common_type_t<T0, T0>`; otherwise there shall be no member `type`.

§ 20.15.7.6
— 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.

(3.3.1) — 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>.

(3.3.2) — [Note: None of the following will apply if there is a specialization common_type<D1, D2>. — end note]

(3.3.3) — Otherwise, if
decay_t<decltype(false ? declval<D1>() : declval<D2>())
denotes a valid type, let C denote that type.

(3.3.4) — Otherwise, if COND_RES(CREF(D1), CREF(D2)) denotes a type, let C denote the type decay_t<COND_RES(CREF(D1), CREF(D2))>.

In either case, the member typedef-name type shall denote the same type, if any, as C. Otherwise, there shall be no member type.

(3.4) — 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 common_type_t<T1, T2>. If there is such a type C, the member typedef-name type shall denote the same type, if any, as common_type_t<C, R>.

Note B: Notwithstanding the provisions of 20.15.2, and pursuant to 16.5.4.2.1, a program may specialize 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 type, but if it does, that member shall be a typedef-name for an accessible and unambiguous cv-unqualified non-reference type T to which each of the types T1 and T2 is explicitly convertible. Moreover, common_type_t<T1, T2> shall denote the same type, if any, as does common_type_t<T2, T1>. No diagnostic is required for a violation of this Note’s rules.

Note C: For the common_reference trait applied to a parameter pack T of types, the member type shall be either defined or not present as follows:

(5.1) — If sizeof...(T) is zero, there shall be no member type.

(5.2) — Otherwise, if sizeof...(T) is one, let T0 denote the sole type in the pack T. The member typedef type shall denote the same type as T0.

(5.3) — Otherwise, if sizeof...(T) is two, let T1 and T2 denote the two types in the pack T. Then

(5.3.1) — If T1 and T2 are reference types and COMMON_REF(T1, T2) is well-formed, then the member typedef type denotes that type.

(5.3.2) — Otherwise, if basic_common_reference<remove_cvref_t<T1>, remove_cvref_t<T2>, XREF(T1), XREF(T2)>>::type is well-formed, then the member typedef type denotes that type.

(5.3.3) — Otherwise, if COND_RES(T1, T2) is well-formed, then the member typedef type denotes that type.

(5.3.4) — Otherwise, if common_type_t<T1, T2> is well-formed, then the member typedef type denotes that type.

(5.3.5) — Otherwise, there shall be no member type.

(5.4) — 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:

(5.4.1) — If there is such a type C, the member typedef type shall denote the same type, if any, as common_reference_t<C, Rest>.

(5.4.2) — 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

§ 20.15.7.6

703
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.

7 \[\text{Example: Given these definitions:}\]
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:

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<PMD, S, char&>, void>);

—end example

20.15.8 Logical operator traits \[\text{[meta.logical]}\]

1 This subclause describes type traits for applying logical operators to other type traits.

template<class... B> struct conjunction : see below {}

2 The class template \texttt{conjunction} forms the logical conjunction of its template type arguments.

3 For a specialization \texttt{conjunction<B_1, ..., B_N>}, if there is a template type argument \(B_i\) for which \texttt{bool(B_i::value)} is \texttt{false}, then instantiating \texttt{conjunction<B_1, ..., B_N>::value} does not require the instantiation of \(B_j::value\) for \(j > i\). \[\text{[Note: This is analogous to the short-circuiting behavior of the built-in operator \&\&]. —end note}\]

4 Every template type argument for which \(B_i::value\) is instantiated shall be usable as a base class and shall have a member \texttt{value} which is convertible to \texttt{bool}, is not hidden, and is unambiguously available in the type.

5 The specialization \texttt{conjunction<B_1, ..., B_N>} has a public and unambiguous base that is either

\[(5.1)\]  
— the first type \(B_i\) in the list \texttt{true_type, B_1, ..., B_N} for which \texttt{bool(B_i::value)} is \texttt{false}, or

\[(5.2)\]  
— if there is no such \(B_i\), the last type in the list.  

\[\text{[Note: This means a specialization of \texttt{conjunction} does not necessarily inherit from either \texttt{true_type} or \texttt{false_type}. —end note]}\]

6 The member names of the base class, other than \texttt{conjunction} and \texttt{operator\&\&}, shall not be hidden and shall be unambiguously available in \texttt{conjunction}.

template<class... B> struct disjunction : see below {}

7 The class template \texttt{disjunction} forms the logical disjunction of its template type arguments.

8 For a specialization \texttt{disjunction<B_1, ..., B_N>}, if there is a template type argument \(B_i\) for which \texttt{bool(B_i::value)} is \texttt{true}, then instantiating \texttt{disjunction<B_1, ..., B_N>::value} does not require the instantiation of \(B_j::value\) for \(j > i\). \[\text{[Note: This is analogous to the short-circuiting behavior of the built-in operator \|\|. —end note]}\]

9 Every template type argument for which \(B_i::value\) is instantiated shall be usable as a base class and shall have a member \texttt{value} which is convertible to \texttt{bool}, is not hidden, and is unambiguously available in the type.

10 The specialization \texttt{disjunction<B_1, ..., B_N>} has a public and unambiguous base that is either
the first type $B_i$ in the list $\text{false_type}$, $B_1$, $\ldots$, $B_N$ for which $\text{bool}(B_i::\text{value})$ is true, or

— if there is no such $B_i$, the last type in the list.

[Note: This means a specialization of \texttt{disjunction} does not necessarily inherit from either \texttt{true_type} or \texttt{false_type}. — end note]

The member names of the base class, other than \texttt{disjunction} and \texttt{operator=}, shall not be hidden and shall be unambiguously available in \texttt{disjunction}.

\begin{verbatim}
template<class B> struct negation : see below {};
\end{verbatim}

The class template \texttt{negation} forms the logical negation of its template type argument. The type \texttt{negation<B>} is a \texttt{Cpp17UnaryTypeTrait} with a base characteristic of \texttt{bool_constant<!bool(B::value)>}.

20.15.9 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.

\begin{verbatim}
enum class endian {
    little = see below,
    big = see below,
    native = see below
};
\end{verbatim}

If all scalar types have size 1 byte, then all of \texttt{endian::little}, \texttt{endian::big}, and \texttt{endian::native} have the same value. Otherwise, \texttt{endian::little} is not equal to \texttt{endian::big}. If all scalar types are big-endian, \texttt{endian::native} is equal to \texttt{endian::big}. If all scalar types are little-endian, \texttt{endian::native} is equal to \texttt{endian::little}. Otherwise, \texttt{endian::native} is not equal to either \texttt{endian::big} or \texttt{endian::little}.

20.15.10 Constant evaluation context

\begin{verbatim}
constexpr bool is_constant_evaluated() noexcept;
\end{verbatim}

Returns: \texttt{true} if and only if evaluation of the call occurs within the evaluation of an expression or conversion that is manifestly constant-evaluated (7.7).

[Example:]
\begin{verbatim}
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{verbatim}

— end example]

20.16 Compile-time rational arithmetic

20.16.1 In general

This subclause describes the ratio library. It provides a class template \texttt{ratio} which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type \texttt{intmax_t}.

Throughout this subclause, the names of template parameters are used to express type requirements. If a template parameter is named \texttt{R1} or \texttt{R2}, and the template argument is not a specialization of the \texttt{ratio} template, the program is ill-formed.
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;
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, 1'000'000'000'000'000'000'000'000'000'000'>; // see below
using zepto = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>; // see below
using atto = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using femto = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using pico = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using nano = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using micro = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using milli = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using centi = ratio<1, 100'000'000'000'000'>;
using deci = ratio<1, 10'000'000'000'000'000'000'000'000'>;
using deca = ratio<10, 1'000'000'000'000'000'000'000'000'000'000'>;
using hecto = ratio<100, 1'000'000'000'000'000'000'000'000'000'000'>;
using kilo = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using mega = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using giga = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using tera = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using peta = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using exa = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>;
using zetta = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>; // see below
using yotta = ratio<1, 1'000'000'000'000'000'000'000'000'000'000'>; // see below
}

namespace std {

// 20.16.3, class template ratio

public:
    static constexpr intmax_t num;
    static constexpr intmax_t den;
    using type = ratio<num, den>;

}
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]

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 $D$:

1. $\text{num}$ shall have the value $\text{sign}(N) \times \text{sign}(D) \times \text{abs}(N) / \text{gcd}$.
2. $\text{den}$ shall have the value $\text{abs}(D) / \text{gcd}$.

### 20.16.4 Arithmetic on ratios

Each of the alias templates `ratio_add`, `ratio_subtract`, `ratio_multiply`, and `ratio_divide` denotes the result of an arithmetic computation on two `ratio` R1 and R2. With X and Y computed (in the absence of arithmetic overflow) as specified by Table 56, 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`.

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.

#### Table 56 — 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><code>ratio_add&lt;R1, R2&gt;</code></td>
<td>$\text{R1}::\text{num} \times \text{R2}::\text{den} + \text{R1}::\text{den} \times \text{R2}::\text{den}$</td>
<td>$\text{R2}::\text{num} \times \text{R1}::\text{den}$</td>
</tr>
<tr>
<td><code>ratio_subtract&lt;R1, R2&gt;</code></td>
<td>$\text{R1}::\text{num} \times \text{R2}::\text{den} - \text{R1}::\text{den} \times \text{R2}::\text{den}$</td>
<td>$\text{R2}::\text{num} \times \text{R1}::\text{den}$</td>
</tr>
<tr>
<td><code>ratio_multiply&lt;R1, R2&gt;</code></td>
<td>$\text{R1}::\text{num} \times \text{R2}::\text{num}$</td>
<td>$\text{R1}::\text{den} \times \text{R2}::\text{den}$</td>
</tr>
<tr>
<td><code>ratio_divide&lt;R1, R2&gt;</code></td>
<td>$\text{R1}::\text{num} \times \text{R2}::\text{den}$</td>
<td>$\text{R1}::\text{den} \times \text{R2}::\text{num}$</td>
</tr>
</tbody>
</table>

[Example:

```cpp
static_assert(ratio_add<ratio<1l, 3l>, ratio<1l, 6l>>::num == 1, "1/3+1/6 == 1/2");
static_assert(ratio_add<ratio<1l, 3l>, ratio<1l, 6l>>::den == 2, "1/3+1/6 == 1/2");
static_assert(ratio_multiply<ratio<1l, 3l>, ratio<3l, 2l>>::num == 1, "1/3*3/2 == 1/2");
static_assert(ratio_multiply<ratio<1l, 3l>, ratio<3l, 2l>>::den == 2, "1/3*3/2 == 1/2");
```

// The following cases may cause the program to be ill-formed under some implementations
```
static_assert(ratio_add<ratio<1l, INT_MAX>, ratio<1l, INT_MAX>>::num == 2,
"1/MAX+1/MAX == 2/MAX");
static_assert(ratio_add<ratio<1l, INT_MAX>, ratio<1l, INT_MAX>>::den == INT_MAX,
"1/MAX+1/MAX == 2/MAX");
static_assert(ratio_multiply<ratio<1l, INT_MAX>, ratio<INT_MAX, 2l>>::num == 1,
"1/MAX * MAX/2 == 1/2");
static_assert(ratio_multiply<ratio<1l, INT_MAX>, ratio<INT_MAX, 2l>>::den == 2,
"1/MAX * MAX/2 == 1/2");
```

— end example]

### 20.16.5 Comparison of ratios

```cpp
template<class R1, class R2>
struct ratio_equal : bool_constant<R1::num == R2::num && R1::den == R2::den> { }

template<class R1, class R2>
struct ratio_not_equal : bool_constant!<ratio_equal_v<R1, R2>> { }
```

§ 20.16.5 707
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
[SI.types]
1
For each of the typedef-names yocto, zepto, zetta, and yotta, if both of the constants used in its specification
are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by
intmax_t, the typedef shall not be defined.

20.17 Class type_index
[type.index]

20.17.1 Header <typeindex> synopsis
[type.index.synopsis]
namespace std {
    class type_index;
    template<class T> struct hash;
    template<> struct hash<type_index>;
}

20.17.2 type_index overview
[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;
        bool 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.
    };
}

1 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.members]
type_index(const type_info& rhs) noexcept;

1 Effects: Constructs a type_index object, the equivalent of target = &rhs.

bool operator==(const type_index& rhs) const noexcept;

2 Returns: *target == *rhs.target.
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).

size_t hash_code() const noexcept;

Returns: target->hash_code().

const char* name() const noexcept;

Returns: target->name().

20.17.4 Hash support

template<> struct hash<type_index>;

For an object index of type type_index, hash<type_index>()(index) shall evaluate to the same result as index.hash_code().

20.18 Execution policies

20.18.1 In general

This subclause 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

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

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> shall be 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 };

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.

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() shall be called.

20.18.5 Parallel execution policy

class execution::parallel_policy { unspecified };

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.

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() shall be called.

20.18.6 Parallel and unsequenced execution policy

class execution::parallel_unsequenced_policy { unspecified };

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.

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() shall be called.

§ 20.18.6
20.18.7 Unsequenced execution policy

The class `execution::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.

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()` shall be called.

20.18.8 Execution policy objects

The header `<execution>` declares global objects associated with each type of execution policy.

20.19 Primitive numeric conversions

20.19.1 Header `<charconv>` synopsis

```cpp
namespace std {
    // floating-point format for primitive numerical conversion
    enum class chars_format {
        scientific = unspecified,
        fixed = unspecified,
        hex = unspecified,
        general = fixed | scientific
    };

    // 20.19.2, primitive numerical output conversion
    struct to_chars_result {
        char* ptr;
        errc ec;
    };

to_chars_result to_chars(char* first, char* last, see below value, int base = 10);

to_chars_result to_chars(char* first, char* last, float value);
to_chars_result to_chars(char* first, char* last, double value);
to_chars_result to_chars(char* first, char* last, long double value);

to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);

to_chars_result to_chars(char* first, char* last, float value, chars_format fmt, int precision);
to_chars_result to_chars(char* first, char* last, double value, chars_format fmt, int precision);
to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt, int precision);

    // 20.19.3, primitive numerical input conversion
    struct from_chars_result {
        const char* ptr;
        errc ec;
    };

    from_chars_result from_chars(const char* first, const char* last, see below value, int base = 10);
```
from_chars_result from_chars(const char* first, const char* last, float& value,
    chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, double& value,
    chars_format fmt = chars_format::general);
from_chars_result from_chars(const char* first, const char* last, long double& value,
    chars_format fmt = chars_format::general);
}

1 The type `chars_format` is a bitmask type (16.4.2.2.4) with elements `scientific`, `fixed`, and `hex`.

### 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.

2 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` (17.3.3.1).

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`.

```c
    to_chars_result to_chars(char* first, char* last, float value);
    to_chars_result to_chars(char* first, char* last, double value);
    to_chars_result to_chars(char* first, char* last, long double value);
```

4 Requires: `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`.

```c
    to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
```

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.

```c
    to_chars_result to_chars(char* first, char* last, float value, chars_format fmt);
    to_chars_result to_chars(char* first, char* last, double value, chars_format fmt);
    to_chars_result to_chars(char* first, char* last, long double value, chars_format fmt);
```

10 Requires: `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.

```c
    to_chars_result to_chars(char* first, char* last, float value,
        chars_format fmt, int precision);
    to_chars_result to_chars(char* first, char* last, double value,
        chars_format fmt, int precision);
```
to_chars_result to_chars(char* first, char* last, long double value,
    chars_format fmt, int precision);

Requires: fmt has the value of one of the enumerators of chars_format.

Effects: value is converted to a string in the style of printf in the "C" locale with the given precision.

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 (17.3.3.1), and the member ec is value-initialized.

from_chars_result from_chars(const char* first, const char* last, see below& value, int base = 10);

Requires: base has a value between 2 and 36 (inclusive).

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.

Throws: Nothing.

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);

Requires: fmt has the value of one of the enumerators of chars_format.

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.

Throws: Nothing.

See also: ISO C 7.22.1.3, 7.22.1.4
21 Strings library

21.1 General

This Clause describes components for manipulating sequences of any non-array trivial standard-layout (6.7) 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 57.

Table 57 — Strings library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>21.3 String classes</td>
<td>&lt;string&gt;</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;cstring&gt; &lt;cwchar&gt; &lt;cstdlib&gt; &lt;cuchar&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 satisfy 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 C 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 58, 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 58 — Character traits requirements

<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::char_type</td>
<td>C</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::int_type</td>
<td>(described in 21.2.2)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::off_type</td>
<td>(described in 29.2.2 and 29.3)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::pos_type</td>
<td>(described in 29.2.2 and 29.3)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::state_type</td>
<td>(described in 21.2.2)</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>X::eq(c,d)</td>
<td>bool</td>
<td>Returns: whether c is to be treated as equal to d.</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 58 — Character traits requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::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>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>Expects: 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],p[i]). 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 58 — Character traits requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X::eof()</code></td>
<td><code>X::int_type</code></td>
<td>Returns: a value e such that <code>X::eq_int_type(e,X::to_int_type(c))</code> is false for all values c.</td>
<td>constant</td>
</tr>
</tbody>
</table>

2 The class template

```cpp
template<class charT> struct char_traits;
```

is provided in the header `<string>` as a basis for explicit specializations.

### 21.2.2 Traits typedefs

**[char.traits.typedefs]**

```cpp
using int_type = see below;
```

1 Expects: `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()`.\(^{225}\)

```cpp
using state_type = see below;
```

2 Expects: `state_type` meets the `Cpp17Destructible` (Table 30), `Cpp17CopyAssignable` (Table 29), `Cpp17CopyConstructible` (Table 27), and `Cpp17DefaultConstructible` (Table 25) requirements.

### 21.2.3 char_traits specializations

**[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;

        static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
        static constexpr bool eq(char_type c1, char_type c2) noexcept;
        static constexpr bool lt(char_type c1, char_type c2) noexcept;
        static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
        static constexpr size_t length(const char_type* s);
        static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
        static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* assign(char_type* s, size_t n, char_type a);
    }
}
```

1 The header `<string>` defines five specializations of the class template `char_traits`: `char_traits<char>`, `char_traits<char8_t>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`.

#### 21.2.3.1 struct char_traits<char>

**[char.traits.specializations.char]**

```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;

        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);
    }
}
```

\(^{225}\) If `eof()` can be held in `char_type` then some iostreams operations can give surprising results.
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 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.

2 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`.

3 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;

    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 char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);
    static constexpr int_type not_eof(int_type c) noexcept;
    static constexpr char_type to_char_type(int_type c) noexcept;
    static constexpr int_type to_int_type(char_type c) noexcept;
    static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
    static constexpr int_type eof() noexcept;
  };
}
```

1 The two-argument members `assign`, `eq`, and `lt` are defined identically to the built-in operators `=`, `==`, and `<` respectively.

2 The member `eof()` returns an implementation-defined constant that cannot appear as a valid UTF-8 code unit.

21.2.3.3 `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;

    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 char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static 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;
  };
}
```

§ 21.2.3.3 717
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>

```cpp
namespace std {
    template<> struct char_traits<char32_t> {
        using char_type = char32_t;
        using int_type = uint_least32_t;
        using off_type = streamoff;
        using pos_type = u32streampos;
        using state_type = mbstate_t;

        static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
        static constexpr bool eq(char_type c1, char_type c2) noexcept;
        static constexpr bool lt(char_type c1, char_type c2) noexcept;
        static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
        static constexpr size_t length(const char_type* s);
        static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
        static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* assign(char_type* s, size_t n, char_type a);
        static constexpr int_type not_eof(int_type c) noexcept;
        static constexpr char_type to_char_type(int_type c) noexcept;
        static constexpr int_type to_int_type(char_type c) noexcept;
        static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
        static constexpr int_type eof() noexcept;
    };
}
```

1. The two-argument members `assign`, `eq`, and `lt` are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

2. The member `eof()` returns an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

### 21.2.3.5 struct char_traits<wchar_t>

```cpp
namespace std {
    template<> struct char_traits<wchar_t> {
        using char_type = wchar_t;
        using int_type = wint_t;
        using off_type = streamoff;
        using pos_type = wstreampos;
        using state_type = mbstate_t;

        static constexpr void assign(char_type& c1, const char_type& c2) noexcept;
        static constexpr bool eq(char_type c1, char_type c2) noexcept;
        static constexpr bool lt(char_type c1, char_type c2) noexcept;
        static constexpr int compare(const char_type* s1, const char_type* s2, size_t n);
        static constexpr size_t length(const char_type* s);
        static constexpr const char_type* find(const char_type* s, size_t n, const char_type& a);
        static constexpr char_type* move(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* copy(char_type* s1, const char_type* s2, size_t n);
        static constexpr char_type* assign(char_type* s, size_t n, char_type a);
        static constexpr int_type not_eof(int_type c) noexcept;
        static constexpr char_type to_char_type(int_type c) noexcept;
        static constexpr int_type to_int_type(char_type c) noexcept;
        static constexpr bool eq_int_type(int_type c1, int_type c2) noexcept;
        static constexpr int_type eof() noexcept;
    };
}
```

1. The two-argument members `assign`, `eq`, and `lt` are defined identically to the built-in operators `=`, `==`, and `<`, respectively.

2. The member `eof()` returns an implementation-defined constant that cannot appear as a Unicode code point.
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 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 <initializer_list>

namespace std {
    // 21.2, character traits
    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>;
    // 21.3.2, basic_string
    template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>
    class basic_string;

    template<class charT, class traits, class Allocator>
    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>
    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>
    basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>&& lhs, basic_string<charT, traits, Allocator>&& rhs);
    template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs, basic_string<charT, traits, Allocator>&& rhs);

§ 21.3.1
template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(const charT* lhs,
              const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(const charT* lhs,
              basic_string<charT, traits, Allocator>&& rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(charT lhs,
              const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(charT lhs,
              basic_string<charT, traits, Allocator>&& rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              const charT* rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              const charT* rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(const basic_string<charT, traits, Allocator>& lhs,
              charT rhs);

template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs,
              charT rhs);

template<class charT, class traits, class Allocator>
    bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                    const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
    bool operator==(const charT* lhs,
                    const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                    const charT* rhs);

template<class charT, class traits, class Allocator>
    bool operator!=(const basic_string<charT, traits, Allocator>& lhs,
                    const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
    bool operator!=(const charT* lhs,
                    const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    bool operator!=(const basic_string<charT, traits, Allocator>& lhs,
                    const charT* rhs);

template<class charT, class traits, class Allocator>
    bool operator< (const basic_string<charT, traits, Allocator>& lhs,
                    const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
    bool operator< (const basic_string<charT, traits, Allocator>&& lhs,
                    const charT* rhs);

template<class charT, class traits, class Allocator>
    bool operator< (const charT* lhs,
                    const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    bool operator< (const charT* lhs,
                    const basic_string<charT, traits, Allocator>&& rhs);

template<class charT, class traits, class Allocator>
    bool operator> (const basic_string<charT, traits, Allocator>& lhs,
                    const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
    bool operator> (const basic_string<charT, traits, Allocator>&& lhs,
                    const charT* rhs);

template<class charT, class traits, class Allocator>
    bool operator> (const charT* lhs,
                    const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
    bool operator> (const charT* lhs,
                    const basic_string<charT, traits, Allocator>&& rhs);
template<class charT, class traits, class Allocator>
bool operator> (const basic_string<charT, traits, Allocator>& lhs,
const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator> (const charT* lhs,
const basic_string<charT, traits, Allocator>& rhs);

// 21.3.3.3, swap
template<class charT, class traits, class Allocator>
void swap(basic_string<charT, traits, Allocator>& lhs,
basic_string<charT, traits, Allocator>& rhs) noexcept(noexcept(lhs.swap(rhs)));

// 21.3.3.4, 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>&
gline(basic_istream<charT, traits>&& is,
const basic_string<charT, traits, Allocator>& str,
charT delim);

// 21.3.3.5, erasure
template<class charT, class traits, class Allocator, class U>
void erase(basic_string<charT, traits, Allocator>& c, const U& value);
template<class charT, class traits, class Allocator, class Predicate>
void erase_if(basic_string<charT, traits, Allocator>& c, Predicate pred);

// basic_string typedef names
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);
long double stold(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(float val);
wstring to_wstring(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>;

§ 21.3.1
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.

```cpp
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_string {
  // types
  using traits_type = traits;
  using value_type = charT;
  using allocator_type = Allocator;
  using size_type = typename allocator_traits<Allocator>::size_type;
  using difference_type = typename allocator_traits<Allocator>::difference_type;
  using pointer = typename allocator_traits<Allocator>::pointer;
  using const_pointer = typename allocator_traits<Allocator>::const_pointer;
  using reference = value_type&;
  using const_reference = const value_type&;

  using iterator = implementation-defined;  // see 22.2
  using const_iterator = implementation-defined;  // see 22.2
  using reverse_iterator = std::reverse_iterator<iterator>;
  using const_reverse_iterator = std::reverse_iterator<const_iterator>;
  static const size_type npos = -1;

  // 21.3.2.2, construct/copy/destroy
  basic_string() noexcept(noexcept(Allocator())) : basic_string(Allocator()) { }
  explicit basic_string(const Allocator& a) noexcept;
  basic_string(const basic_string& str);
  basic_string(basic_string&& str) noexcept;
  basic_string(const basic_string& str, size_type pos, const Allocator& a = Allocator());
  basic_string(const basic_string& str, size_type pos, size_type n,
               const Allocator& a = Allocator());
  template<class T>
  basic_string(const T& t, size_type pos, size_type n, const Allocator& a = Allocator());
  template<class T>
  explicit basic_string(const T& t, const Allocator& a = Allocator());
  basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

§ 21.3.2 723
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
    basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
basic_string(const basic_string&, const Allocator&);
basic_string(basic_string&&, const Allocator&);

~basic_string();
basic_string& operator=(const basic_string& str);
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>
    basic_string& operator=(const T& t);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(initializer_list<charT>);

// 21.3.2.3, 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;

// 21.3.2.4, capacity
size_type size() const noexcept;
size_type length() const noexcept;
size_type max_size() const noexcept;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const noexcept;
void reserve(size_type res_arg);
void shrink_to_fit();
void clear() noexcept;
[[nodiscard]] bool empty() const noexcept;

// 21.3.2.5, element access
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

const charT& front() const;
charT& front();
const charT& back() const;
charT& back();

// 21.3.2.6, modifiers
basic_string& operator+=(const basic_string& str);
template<T>
    basic_string& operator+=(const T& t);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& operator+=(initializer_list<charT>);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
  basic_string& append(const T& t);
template<class T>
  basic_string& append(const T& t, size_type pos, size_type n = npos);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(initializer_list<charT>);

void push_back(charT c);

basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
             allocator_traits<Allocator>::is_always_equal::value);
basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);
template<class T>
  basic_string& assign(const T& t);
template<class T>
  basic_string& assign(const T& t, size_type pos, size_type n = npos);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);

basic_string& insert(size_type pos, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str, size_type pos2, size_type n = npos);
template<class T>
  basic_string& insert(size_type pos, const T& t);
template<class T>
  basic_string& insert(size_type pos1, const T& t, size_type pos2, size_type n = npos);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  iterator insert(const_iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);

void pop_back();

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
  size_type pos2, size_type n2 = npos);
template<class T>
  basic_string& replace(size_type pos1, size_type n1, const T& t);
template<class T>
  basic_string& replace(size_type pos1, size_type n1, const T& t,
    size_type pos2, size_type n2 = npos);
basic_string& replace(size_type pos, size_type ni, const charT* s, size_type n2);
basic_string& replace(size_type pos, size_type ni, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);

basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);

template<class T>
  basic_string& replace(const_iterator i1, const_iterator i2, const T& t);

basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);

basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);

basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);

template<class InputIterator>
  basic_string& replace(const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2);

basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT>);

size_type copy(charT* s, size_type n, size_type pos = 0) const;

void swap(basic_string& str)
  noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
      allocator_traits<Allocator>::is_always_equal::value);

// 21.3.2.7, string operations
const charT* c_str() const noexcept;

const charT* data() const noexcept;

charT* data() noexcept;

operator basic_string_view<charT, traits>() const noexcept;

allocator_type get_allocator() const noexcept;

template<class T>
  size_type find (const T& t, size_type pos = 0) const noexcept(see below);

size_type find (const basic_string& str, size_type pos = 0) const noexcept;

size_type find (const charT* s, size_type pos, size_type n) const;

size_type find (const charT* s, size_type pos = 0) const;

size_type find (charT c, size_type pos = 0) const noexcept;

template<class T>
  size_type rfind(const T& t, size_type pos = npos) const noexcept(see below);

size_type rfind(const basic_string& str, size_type pos = npos) const noexcept;

size_type rfind(const charT* s, size_type pos, size_type n) const;

size_type rfind(const charT* s, size_type pos = npos) const;

size_type rfind(charT c, size_type pos = npos) const noexcept;

template<class T>
  size_type find_first_of(const T& t, size_type pos = 0) const noexcept(see below);

size_type find_first_of(const basic_string& str, size_type pos = 0) const noexcept;

size_type find_first_of(const charT* s, size_type pos, size_type n) const;

size_type find_first_of(const charT* s, size_type pos = 0) const;

size_type find_first_of(charT c, size_type pos = 0) const noexcept;

template<class T>
  size_type find_last_of (const T& t, size_type pos = npos) const noexcept(see below);

size_type find_last_of (const basic_string& str, size_type pos = npos) const noexcept;

size_type find_last_of (const charT* s, size_type pos, size_type n) const;

size_type find_last_of (const charT* s, size_type pos = npos) const;

size_type find_last_of (charT c, size_type pos = npos) const noexcept;

template<class T>
  size_type find_first_not_of(const T& t, size_type pos = 0) const noexcept(see below);

size_type find_first_not_of(const basic_string& str, size_type pos = 0) const noexcept;

size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;

size_type find_first_not_of(const charT* s, size_type pos = 0) const;

size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;

template<class T>
  size_type find_last_not_of (const T& t, size_type pos = npos) const noexcept(see below);

size_type find_last_not_of (const basic_string& str, size_type pos = npos) const noexcept;

size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;

size_type find_last_not_of (const charT* s, size_type pos = npos) const;

size_type find_last_not_of (charT c, size_type pos = npos) const noexcept;

§ 21.3.2 726
basic_string substr(size_type pos = 0, size_type n = npos) const;
    template<class T>
    int compare(const T& t) const noexcept(see below);
    template<class T>
    int compare(size_type pos1, size_type n1, const T& t) const;
    template<class T>
    int compare(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos) const;
    int compare(const basic_string& str) const noexcept;
    template<class T>
    int compare(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos) const;
    int compare(const charT* s) const;
    int compare(size_type pos1, size_type n1, const charT* s) const;
    int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

bool starts_with(basic_string_view<charT, traits> x) const noexcept;
bool starts_with(charT x) const noexcept;
bool starts_with(const charT* x) const;
bool ends_with(basic_string_view<charT, traits> x) const noexcept;
bool ends_with(charT x) const noexcept;
bool ends_with(const charT* x) const;
}

4 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

1 If any operation would cause size() to exceed max_size(), that operation throws an exception object of type length_error.

2 If any member function or operator of basic_string throws an exception, that function or operator has no other effect on the basic_string object.

3 In every specialization basic_string<charT, traits, Allocator>, the type allocator_traits<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 satisfy 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]

§ 21.3.2.1
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:

- Passing as an argument to any standard library function taking a reference to non-const `basic_string` as an argument.\(^{226}\)
- Calling non-const member functions, except `operator[]`, `at`, `data`, `front`, `back`, `begin`, `rbegin`, `end`, and `rend`.

### 21.3.2.2 Constructors and assignment operators

```cpp
explicit basic_string(const Allocator& a) noexcept;

basic_string(const basic_string& str);
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.

basic_string(const basic_string& str, size_type pos, const Allocator& a = Allocator());
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>
    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>
    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)`.

basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
Expects: `[s, s + n)` is a valid range.
Effects: Constructs an object whose initial value is the range `[s, s + n)`.
Ensures: `size()` is equal to `n`, and `traits::compare(data(), s, n)` is equal to 0.

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)`.

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`.

---

\(^{226}\) 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()`.
template<class InputIterator>
    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 66.

basic_string(initializer_list<charT> il, const Allocator& a = Allocator());

Effects: Equivalent to basic_string(il.begin(), il.end(), a).

basic_string(const basic_string& str, const Allocator& alloc);

Effects: Constructs an object whose value is that of str prior to this call. The stored allocator is
constructed from alloc. In the second form, str is left in a valid but unspecified state.

Throws: The second form throws nothing if alloc == str.get_allocator().

template<class InputIterator, class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
    basic_string(InputIterator, InputIterator, Allocator = Allocator())
-> basic_string<typename iterator_traits<InputIterator>::value_type,
    char_traits<typename iterator_traits<InputIterator>::value_type>,
    Allocator>;

Constraints: InputIterator is a type that qualifies as an input iterator, and Allocator is a type that
qualifies as an allocator (22.2.1).

template<class charT, class traits, class Allocator = allocator<charT>>
    explicit basic_string(basic_string_view<charT, traits>, const Allocator& = Allocator())
-> basic_string<charT, traits, Allocator>;

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).

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.

basic_string& operator=(basic_string& str)

noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
    allocator_traits<Allocator>::is_always_equal::value);

Effects: Move assigns as a sequence container (22.2), except that iterators, pointers and references may
be invalidated.

Returns: *this.

template<class T>
    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);

basic_string& operator=(const charT* s);

Effects: Equivalent to: return *this = basic_string_view<charT, traits>(s);

basic_string& operator=(charT c);

Effects: Equivalent to:
return *this = basic_string_view<charT, traits>(addressof(c), 1);

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 Iterator support

iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;

Returns: An iterator referring to the first character in the string.

iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;

Returns: An iterator which is the past-the-end value.

reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;

Returns: An iterator which is semantically equivalent to reverse_iterator(end()).

reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;

Returns: An iterator which is semantically equivalent to reverse_iterator(begin()).

21.3.2.4 Capacity

size_type size() const noexcept;
size_type length() const noexcept;

Returns: A count of the number of char-like objects currently in the string.

Complexity: Constant time.

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.

void resize(size_type n, charT c);

Effects: Alters the value of *this as follows:

(5.1) If n <= size(), erases the last size() - n elements.
(5.2) If n > size(), appends n - size() copies of c.

void resize(size_type n);

Effects: Equivalent to resize(n, charT()).

size_type capacity() const noexcept;

Returns: The size of the allocated storage in the string.

Complexity: Constant time.
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.

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]

void clear() noexcept;

Effects: Equivalent to: erase(begin(), end());

[[nodiscard]] bool empty() const noexcept;

Effects: Equivalent to: return size() == 0;

21.3.2.5 Element access [string.access]

const_reference operator[](size_type pos) const;
reference operator[](size_type pos);

Expects: 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.

const_reference at(size_type pos) const;
reference at(size_type pos);

Throws: out_of_range if pos >= size().
Returns: operator[](pos).

const charT& front() const;
charT& front();

Expects: !empty().
Effects: Equivalent to: return operator[](0);

const charT& back() const;
charT& back();

Expects: !empty().
Effects: Equivalent to: return operator[](size() - 1);

21.3.2.6 Modifiers [string.modifiers]

21.3.2.6.1 basic_string::operator+= [string.op.append]

basic_string& operator+=(const basic_string& str);

Effects: Equivalent to: return append(str);
template<class T>
    basic_string& operator+=(const T& t);

2 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.

3 Effects: Equivalent to:
    basic_string_view<charT, traits> sv = t;
    return append(sv);

basic_string& operator+=(const charT* s);
4 Effects: Equivalent to: return append(s);

basic_string& operator+=(charT c);
5 Effects: Equivalent to: return append(size_type{1}, c);

basic_string& operator+=(initializer_list<charT> il);
6 Effects: Equivalent to: return append(il);

21.3.2.6.2 basic_string::append [string.append]

basic_string& append(const basic_string& str);
1 Effects: Equivalent to: return append(str.data(), str.size());

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>
    basic_string& append(const T& t);
3 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.

4 Effects: Equivalent to:
    basic_string_view<charT, traits> sv = t;
    return append(sv.data(), sv.size());

template<class T>
    basic_string& append(const T& t, size_type pos, size_type n = npos);
5 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.

6 Effects: Equivalent to:
    basic_string_view<charT, traits> sv = t;
    return append(sv.substr(pos, n));

basic_string& append(const charT* s, size_type n);
7 Expects: [s, s + n) is a valid range.
8 Effects: Appends a copy of the range [s, s + n) to the string.
9 Returns: *this.

basic_string& append(const charT* s);
10 Effects: Equivalent to: return append(s, traits::length(s));
basic_string& append(size_type n, charT c);
  
  Effects: Appends n copies of c to the string.
  
  Returns: *this.

template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
  
  Constraints: InputIterator is a type that qualifies as an input iterator (22.2.1).
  
  Effects: Equivalent to: return append(basic_string(first, last, get_allocator()));

basic_string& append(initializer_list<charT> il);
  
  Effects: Equivalent to: return append(il.begin(), il.size());

void push_back(charT c);
  
  Effects: Equivalent to append(size_type{1}, c).

21.3.2.6.3  basic_string::assign  [string.assign]

basic_string& assign(const basic_string& str);
  
  Effects: Equivalent to: return *this = str;

basic_string& assign(basic_string&& str)
  
  noexcept(allocation_traits<Allocator>::propagate_on_container_move_assignment::value ||
            allocation_traits<Allocator>::is_always_equal::value);
  
  Effects: Equivalent to: return *this = std::move(str);

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>
  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>
  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));

basic_string& assign(const charT* s, size_type n);
  
  Expects: [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.

basic_string& assign(const charT* s);
  
  Effects: Equivalent to: return assign(s, traits::length(s));
basic_string& assign(initializer_list<charT> il);
12  Effects: Equivalent to:
13     return assign(il.begin(), il.size());

basic_string& assign(size_type n, charT c);
14  Effects: Equivalent to:
15     clear();
16     resize(n, c);
17     return *this;

template<class InputIterator>
18     basic_string& assign(InputIterator first, InputIterator last);
19  Constraints: InputIterator is a type that qualifies as an input iterator (22.2.1).
20  Effects: Equivalent to: return assign(basic_string(first, last, get_allocator()));

21.3.2.6.4 basic_string::insert
22 [string.insert]

basic_string& insert(size_type pos, const basic_string& str);
1  Effects: Equivalent to: return insert(pos, str.data(), str.size());

basic_string& insert(size_type pos1, const basic_string& str, size_type pos2, size_type n = npos);
2  Effects: Equivalent to:
3     return insert(pos1, basic_string_view<charT, traits>(str), pos2, n);

template<class T>
4     basic_string& insert(size_type pos, const T& t);
5  Constraints:
6     — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
7     — is_convertible_v<const T&, const charT*> is false.
8  Effects: Equivalent to:
9     basic_string_view<charT, traits> sv = t;
10    return insert(pos, sv.data(), sv.size());

template<class T>
11     basic_string& insert(size_type pos1, const T& t, size_type pos2, size_type n = npos);
12  Constraints:
13     — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
14     — is_convertible_v<const T&, const charT*> is false.
15  Effects: Equivalent to:
16     basic_string_view<charT, traits> sv = t;
17     return insert(pos1, sv.substr(pos2, n));

basic_string& insert(size_type pos, const charT* s, size_type n);
18  Expects: [s, s + n) is a valid range.
19  Throws:
20     — out_of_range if pos > size(),
21     — length_error if n > max_size() - size(), or
22     — any exceptions thrown by allocator_traits<Allocator>::allocate.
23  Effects: Inserts a copy of the range [s, s + n) immediately before the character at position pos if
24     pos < size(), or otherwise at the end of the string.
25  Returns: *this.

basic_string& insert(size_type pos, const charT* s);
26  Effects: Equivalent to: return insert(pos, s, traits::length(s));
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:*
- out_of_range if pos > size(),
- length_error if n > max_size() - size(), or
- any exceptions thrown by allocator_traits<Allocator>::allocate.

iterator insert(const_iterator p, charT c);

*Expects: 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.

iterator insert(const_iterator p, size_type n, charT c);

*Expects: 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>
iterator insert(const_iterator p, InputIterator first, InputIterator last);

*Constraints: InputIterator is a type that qualifies as an input iterator (22.2.1).
*Expects: 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.

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

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.

iterator erase(const_iterator p);

*Expects: 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.

iterator erase(const_iterator first, const_iterator last);

*Expects: 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.

§ 21.3.2.6.5 735
void pop_back();

Expects: !empty().

Throws: Nothing.

Effects: Equivalent to erase(end() - 1).

21.3.2.6.6  basic_string::replace

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);

Effects: Equivalent to: return replace(pos1, n1, str.data(), str.size());

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>
basic_string& replace(size_type pos1, size_type n1, 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 replace(pos1, n1, sv.data(), sv.size());

template<class T>
basic_string& replace(size_type pos1, size_type n1, const T& t,
size_type pos2, size_type n2 = 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 replace(pos1, n1, sv.substr(pos2, n2));

basic_string& replace(size_type pos1, size_type n1, const charT* s, size_type n2);

Expects: [s, s + n2) is a valid range.

Throws:
(8.1) — out_of_range if pos1 > size(),
(8.2) — length_error if the length of the resulting string would exceed max_size() (see below), or
(8.3) — any exceptions thrown by allocator_traits<Allocator>::allocate.

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.

basic_string& replace(size_type pos, size_type n, const charT* s);

Effects: Equivalent to: return replace(pos, n, s, traits::length(s));

basic_string& replace(size_type pos1, size_type n1, size_type n2, charT c);

Throws:
(12.1) — out_of_range if pos1 > size(),
(12.2) — length_error if the length of the resulting string would exceed max_size() (see below), or
any exceptions thrown by \texttt{allocator_traits<Allocator>::allocate}.

\textbf{Effects:} Determines the effective length $xlen$ of the string to be removed as the smaller of $n1$ and $\text{size()} - \text{pos1}$. If $\text{size()} - xlen \geq \max\text{size()} - n2$ throws $\text{length}\_\text{error}$. Otherwise, the function replaces the characters in the range $[\text{begin()} + \text{pos1}, \text{begin()} + \text{pos1} + xlen]$ with $n2$ copies of $c$.

\textbf{Returns:} $*\text{this}$.

\begin{verbatim}
basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
\end{verbatim}

\textbf{Effects:} Equivalent to: \begin{verbatim}
return replace(i1, i2, basic_string_view<charT, traits>(str));
\end{verbatim}

\begin{verbatim}
template<class T>
basic_string& replace(const_iterator i1, const_iterator i2, const T& t);
\end{verbatim}

\textbf{Constraints:}

\begin{enumerate}
\item is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
\item is_convertible_v<const T&, const charT*> is false.
\end{enumerate}

\textbf{Expects:} $[\text{begin()}, \text{i1})$ and $[\text{i1}, \text{i2})$ are valid ranges.

\textbf{Effects:} Equivalent to:

\begin{verbatim}
basic_string_view<charT, traits> sv = t;
return replace(i1 - begin(), i2 - i1, sv.data(), sv.size());
\end{verbatim}

\begin{verbatim}
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);
\end{verbatim}

\textbf{Effects:} Equivalent to:

\begin{verbatim}
return replace(i1, i2, basic_string_view<charT, traits>(s, n));
\end{verbatim}

\begin{verbatim}
basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);
\end{verbatim}

\textbf{Effects:} Equivalent to:

\begin{verbatim}
return replace(i1, i2, basic_string_view<charT, traits>(s));
\end{verbatim}

\begin{verbatim}
basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);
\end{verbatim}

\textbf{Expects:} $[\text{begin()}, \text{i1})$ and $[\text{i1}, \text{i2})$ are valid ranges.

\textbf{Effects:} Equivalent to:

\begin{verbatim}
return replace(i1 - begin(), i2 - i1, n, c);
\end{verbatim}

\begin{verbatim}
template<class InputIterator>
basic_string& replace(const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2);
\end{verbatim}

\textbf{Constraints:} \texttt{InputIterator} is a type that qualifies as an input iterator (\ref{22.2.1}).

\textbf{Effects:} Equivalent to:

\begin{verbatim}
return replace(i1, i2, basic_string(j1, j2, get_allocator()));
\end{verbatim}

\begin{verbatim}
basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);
\end{verbatim}

\textbf{Effects:} Equivalent to:

\begin{verbatim}
return replace(i1, i2, il.begin(), il.size());
\end{verbatim}

\begin{verbatim}

21.3.2.6.7 basic_string::copy
size_type copy(charT* s, size_type n, size_type pos = 0) const;
\end{verbatim}

\textbf{Effects:} Equivalent to:

\begin{verbatim}
return basic_string_view<charT, traits>(*this).copy(s, n, pos);
\end{verbatim}

\textbf{Note:} This does not terminate $s$ with a null object. \textit{— end note}

\begin{verbatim}
21.3.2.6.8 basic_string::swap
void swap(basic_string& s)
\end{verbatim}

\textbf{noexcept\texttt{\{}\texttt{allocator_traits\textless\textit{Allocator}\textgreater\texttt::propagate\_on\_container\_swap\texttt::value ||
\texttt{allocator_traits\textless\textit{Allocator}\textgreater\texttt::is\_always\_equal\texttt::value\texttt\}}} \texttt{\}}

\textbf{Expects:} \texttt{allocator_traits<Allocator>::propagate_on_container_swap::value} is true or \texttt{get\_allocator() == s.get\_allocator()}.

\textbf{Ensures:} $*\text{this}$ contains the same sequence of characters that was in $s$, $s$ contains the same sequence of characters that was in $*\text{this}$.

\textbf{Throws:} Nothing.

\textbf{Complexity:} Constant time.
21.3.2.7 String operations

21.3.2.7.1 Accessors

const charT* c_str() const noexcept;
const charT* data() const noexcept;

1 Returns: A pointer p such that p + i == addressof(operator[](i)) for each i in [0, size()].
2 Complexity: Constant time.
3 Remarks: The program shall not modify any of the values stored in the character array; otherwise, the behavior is undefined.

charT* data() noexcept;

4 Returns: A pointer p such that p + i == addressof(operator[](i)) for each i in [0, size()].
5 Complexity: Constant time.
6 Remarks: The program shall not modify the value stored at p + size() to any value other than charT(); otherwise, the behavior is undefined.

operator basic_string_view<charT, traits>() const noexcept;

7 Effects: Equivalent to: return basic_string_view<charT, traits>(data(), size());

allocator_type get_allocator() const noexcept;

8 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 find, rfind, find_first_of, find_last_of, find_first_not_of, and find_last_not_of.

(1.1) — Each member function of the form

size_type F(const basic_string& str, size_type pos) const noexcept;

has effects equivalent to: return F(basic_string_view<charT, traits>(str), pos);

(1.2) — Each member function of the form

size_type F(const charT* s, size_type pos) const;

has effects equivalent to: return F(basic_string_view<charT, traits>(s), pos);

(1.3) — Each member function of the form

size_type F(const charT* s, size_type pos, size_type n) const;

has effects equivalent to: return F(basic_string_view<charT, traits>(s, n), pos);

(1.4) — Each member function of the form

size_type F(charT c, size_type pos) const noexcept;

has effects equivalent to:

return F(basic_string_view<charT, traits>(addressof(c), 1), pos);

template<class T>
size_type find(const T& t, size_type pos = 0) const noexcept;

template<class T>
size_type rfind(const T& t, size_type pos = npos) const noexcept;

template<class T>
size_type find_first_of(const T& t, size_type pos = 0) const noexcept;

template<class T>
size_type find_last_of(const T& t, size_type pos = npos) const noexcept;

template<class T>
size_type find_first_not_of(const T& t, size_type pos = 0) const noexcept;

template<class T>
size_type find_last_not_of(const T& t, size_type pos = npos) const noexcept;

Constraints:

(2.1) — is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
— `is_convertible_v<const T&, const charT*>` is false.

*Effects:* Let \( G \) be the name of the function. Equivalent to:

```cpp
define s = *this, sv = t;
define return \( s.G(sv, \text{pos}) \);
```

*Remarks:* The expression inside `noexcept` is equivalent to `is_nothrow_convertible_v<const T&, basic_string_view<charT, traits>>`.

### 21.3.2.7.3 `basic_string::substr`  
[string.substr]

```cpp
define basic_string substr(size_type pos = 0, size_type n = npos) const;
```

*Throws:* `out_of_range` if `pos > size()`.

*Effects:* Determines the effective length `rlen` of the string to copy as the smaller of `n` and `size() - pos`.

*Returns:* `basic_string(data() + pos, rlen)`.

### 21.3.2.7.4 `basic_string::compare`  
[string.compare]

```cpp
template<class T>
define int compare(const T& t) const noexcept;
```

*Constraints:*

1. `is_convertible_v<const T&, basic_string_view<charT, traits>>` is true and
2. `is_convertible_v<const T&, const charT*>` is false.

*Effects:* Equivalent to: `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>>`.

```cpp
template<class T>
define int compare(size_type pos1, size_type n1, const T& t) const;
```

*Constraints:*

1. `is_convertible_v<const T&, basic_string_view<charT, traits>>` is true and
2. `is_convertible_v<const T&, const charT*>` is false.

*Effects:* Equivalent to: `return basic_string_view<charT, traits>(*this).substr(pos1, n1).compare(t)`;

```cpp
template<class T>
define int compare(size_type pos1, size_type n1, const basic_string& str) const;
```

*Effects:* Equivalent to: `return compare(basic_string_view<charT, traits>(str))`;

```cpp
define int compare(size_type pos1, size_type n1, const basic_string& str) const;
```

*Effects:* Equivalent to: `return compare(pos1, n1, basic_string_view<charT, traits>(str))`;

```cpp
define int compare(size_type pos1, size_type n1, size_type pos2, size_type n2 = npos) const;
```

*Effects:* Equivalent to: `return compare(pos1, n1, basic_string_view<charT, traits>(str), pos2, n2)`;
int compare(const charT* s) const;  
**Effects:** Equivalent to: return compare(basic_string_view<charT, traits>(s));

int compare(size_type pos, size_type n1, const charT* s) const;  
**Effects:** Equivalent to: return compare(pos, n1, basic_string_view<charT, traits>(s));

int compare(size_type pos, size_type n1, const charT* s, size_type n2) const;  
**Effects:** Equivalent to: return compare(pos, n1, basic_string_view<charT, traits>(s, n2));

21.3.2.7.5 basic_string::starts_with  
**[string.starts.with]**

bool starts_with(basic_string_view<charT, traits> x) const noexcept;  
bool starts_with(charT x) const noexcept;  
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  
**[string.ends.with]**

bool ends_with(basic_string_view<charT, traits> x) const noexcept;  
bool ends_with(charT x) const noexcept;  
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  
**[string.nonmembers]**

21.3.3.1 operator+  
**[string.op.plus]**

template<class charT, class traits, class Allocator>

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>

basic_string<charT, traits, Allocator>

operator+(const charT* lhs,  
const charT* rhs);

template<class charT, class traits, class Allocator>

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>

basic_string<charT, traits, Allocator>

operator+(basic_string<charT, traits, Allocator>&& lhs,  
basic_string<charT, traits, Allocator>&& rhs);

**Effects:** Equivalent to:

lhs.append(rhs);  
return std::move(lhs);

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>

operator+(basic_string<charT, traits, Allocator>&& lhs,  
basic_string<charT, traits, Allocator>&& rhs);

**Effects:** Equivalent to:

lhs.append(rhs);  
return std::move(lhs);
except that both \texttt{lhs} and \texttt{rhs} are left in valid but unspecified states. [Note: If \texttt{lhs} and \texttt{rhs} have equal allocators, the implementation may move from either. — end note]

\begin{verbatim}
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator> 
operator+(const basic_string<charT, traits, Allocator>&\& lhs, 
basic_string<charT, traits, Allocator>&\& rhs);

template<class charT, class traits, class Allocator> 
basic_string<charT, traits, Allocator> 
operator+(const charT* lhs, basic_string<charT, traits, Allocator>&\& rhs);

Effects: Equivalent to:
          rhs.insert(0, lhs);
          return std::move(rhs);

template<class charT, class traits, class Allocator> 
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> 
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> 
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> 
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> 
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);
\end{verbatim}

21.3.3.2 Non-member comparison functions

\begin{verbatim}
template<class charT, class traits, class Allocator>
bool operator==(const basic_string<charT, traits, Allocator>&\& lhs, 
const basic_string<charT, traits, Allocator>&\& rhs) noexcept;

template<class charT, class traits, class Allocator> 
bool operator==(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);
\end{verbatim}
template<class charT, class traits, class Allocator>
bool operator==(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator!=(const basic_string<charT, traits, Allocator>& lhs, const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
bool operator!=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator!=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator<(const basic_string<charT, traits, Allocator>& lhs, const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
bool operator<(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator<(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator>(const basic_string<charT, traits, Allocator>& lhs, const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
bool operator>(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator>(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator<=(const basic_string<charT, traits, Allocator>& lhs, const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
bool operator<=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator<=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT, traits, Allocator>& lhs, const basic_string<charT, traits, Allocator>& rhs) noexcept;

template<class charT, class traits, class Allocator>
bool operator>=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

template<class charT, class traits, class Allocator>
bool operator>=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

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>
void swap(basic_string<charT, traits, Allocator>& lhs, basic_string<charT, traits, Allocator>& rhs) noexcept(noexcept(lhs.swap(rhs)));

Effects: Equivalent to lhs.swap(rhs).

21.3.3.4 Inserters and extractors

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:
(1.1) n characters are stored;
(1.2) end-of-file occurs on the input sequence;
(1.3) 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::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);

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:

(6.1) end-of-file occurs on the input sequence (in which case, the getline function calls is.setstate(ios_base::eofbit));
(6.2) traits::eq(c, delim) for the next available input character c (in which case, c is extracted but not appended) (29.5.5.4)
(6.3) 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).

Returns: is.

template<class charT, class traits, class Allocator, class U>
void erase(basic_string<charT, traits, Allocator>& c, const U& value);

Effects: Equivalent to: c.erase(remove(c.begin(), c.end(), value), c.end());

21.3.3.5 Erasure [string.eraser]

§ 21.3.3.5
template<class charT, class traits, class Allocator, class Predicate>
void erase_if(basic_string<charT, traits, Allocator>& c, Predicate pred);

Effects: Equivalent to: c.erase(remove_if(c.begin(), c.end(), pred), c.end());

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
strtoul(str.c_str(), ptr, base), strtoll(str.c_str(), ptr, base), and strtoull(str.c_str(), ptr, base), respectively. Each function returns the converted result, if any. The argument
ptr designates a pointer to an object internal to the function that is used to determine what to store
at *idx. If the function does not throw an exception and idx != 0, the function stores in *idx the
index of the first unconverted element of str.

Returns: The converted result.

Throws: invalid_argument if strtol, strtoul, strtell, or strtoull reports that no conversion
could be performed. Throws out_of_range if strtol, strtoul, strtell or strtoull sets errno to
ERANGE, or if the converted value is outside the range of representable values for the return type.

float stof(const string& str, size_t* idx = nullptr);
double stod(const string& str, size_t* idx = nullptr);
long double stold(const string& str, size_t* idx = nullptr);

Effects: These functions call strtof(str.c_str(), ptr), strtod(str.c_str(), ptr), and strtold(
str.c_str(), ptr), respectively. Each function returns the converted result, if any. The argument
ptr designates a pointer to an object internal to the function that is used to determine what to store
at *idx. If the function does not throw an exception and idx != 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
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 wcstol, wcstoul, wcstoll, or wcstoull reports that no conversion could be performed. Throws out_of_range if the converted value is outside the range of representable values for the return type.

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.

Throws: invalid_argument if wcstof, wcstod, or wcstold reports that no conversion could be performed. Throws out_of_range if wcstof, wcstod, or wcstold sets errno to ERANGE.

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"%lld", L"%lu", L"%llu", L"%f", L"%f", or L"%Lf", respectively, where buf designates an internal character buffer of sufficient size buffsz.

21.3.5 Hash support

21.3.6 Suffix for basic_string literals
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`.

[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

```cpp
namespace std {
    // 21.4.2, class template basic_string_view
    template<class charT, class traits = char_traits<charT>>
    class basic_string_view;

    // 21.4.3, 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 bool operator!=(basic_string_view<charT, traits> x,
                               basic_string_view<charT, traits> y) noexcept;

    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 bool operator> (basic_string_view<charT, traits> x,
                               basic_string_view<charT, traits> y) noexcept;

    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 bool operator>=(basic_string_view<charT, traits> x,
                               basic_string_view<charT, traits> y) noexcept;

    // see 21.4.3, sufficient additional overloads of comparison functions

    // 21.4.4, inserters and extractors
    template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os,
                   basic_string_view<charT, traits> str);

    // basic_string_view typedef names
    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>;
}
```
// 21.4.5, 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 {
// 21.4.6, suffix for basic_string_view literals
constexpr string_view operator"sv(const char* str, size_t len) noexcept;
constexpr u8string_view operator"sv(const char8_t* str, size_t len) noexcept;
constexpr u16string_view operator"sv(const char16_t* str, size_t len) noexcept;
constexpr u32string_view operator"sv(const char32_t* str, size_t len) noexcept;
constexpr wstring_view operator"sv(const wchar_t* 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

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;
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& sv) noexcept = default;
constexpr basic_string_view(const basic_string_view&& sv) noexcept = default;
constexpr basic_string_view(const charT* str);
constexpr basic_string_view(const charT* str, size_type len);

// 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;

friend constexpr const_iterator begin(basic_string_view sv) noexcept { return sv.begin(); }
friend constexpr const_iterator end(basic_string_view sv) noexcept { return sv.end(); }

227) Because basic_string_view refers to a constant sequence, iterator and const_iterator are the same type.
// 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* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type rfind(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind(const charT* s, size_type pos = npos) const;
constexpr size_type find_first_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(charT c, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(charT c, size_type pos, size_type n) const;
constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(charT c, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;
constexpr size_type find_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find_first_not_of(charT c, size_type pos = 0) const;
constexpr size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_not_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_not_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;
constexpr size_type find_last_not_of(const charT* s, size_type pos = 0) const;
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, size_type n) const;
constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;

private:
    const_pointer data_; // exposition only
    size_type size_;     // exposition only
};

1 In every specialization basic_string_view<charT, traits>, the type traits shall satisfy 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 The type iterator satisfies the constexpr iterator requirements (23.3.1).

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;
1 Ensures: size_ == 0 and data_ == nullptr.

constexpr basic_string_view(const charT* str);
2 Expects: [str, str + traits::length(str)) is a valid range.
3 Effects: Constructs a basic_string_view, initializing data_ with str and initializing size_ with
   traits::length(str).
4 Complexity: $O(\text{traits::length(str)})$.

constexpr basic_string_view(const charT* str, size_type len);
5 Expects: [str, str + len) is a valid range.
6 Effects: Constructs a basic_string_view, initializing data_ with str and initializing size_ with len.

21.4.2.2 Iterator support [string.view.iterators]

using const_iterator = implementation-defined;
1 A type that meets the requirements of a constant Cpp17RandomAccessIterator (23.3.5.6) and models
   ContiguousIterator (23.3.4.14), whose value_type is the template parameter charT.
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 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;
4 Returns: An iterator such that
   
   (4.1) if !empty(), addressof(*begin()) == data_,
   (4.2) otherwise, an unspecified value such that [begin(), end()) is a valid range.

constexpr const_iterator end() const noexcept;
constexpr const_iterator cend() const noexcept;
5 Returns: begin() + size().

constexpr const_reverse_iterator rbegin() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
6 Returns: const_reverse_iterator(end()).

constexpr const_reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;
7 Returns: const_reverse_iterator(begin()).

§ 21.4.2.2 749
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;

Expects: 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;

Expects: !empty().
Returns: data_[0].
Throws: Nothing.

constexpr const_reference back() const;

Expects: !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);

Expects: n <= size().
Effects: Equivalent to: data_ += n; size_ -= n;

constexpr void remove_suffix(size_type n);

Expects: 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().

Expects: [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 size() and str.size().

Effects: Determines rlen, the effective length of the strings to compare. The function then compares
the two strings by calling traits::compare(data(), str.data(), rlen).

Complexity: O(rlen).

Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as
indicated in Table 59.
```

```
Table 59 — compare() results

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size() &lt; str.size()</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>size() == str.size()</td>
<td>0</td>
</tr>
<tr>
<td>size() &gt; str.size()</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>
```

```
constexpr int compare(size_type pos1, size_type n1, basic_string_view str) const;

Effects: Equivalent to: return substr(pos1, n1).compare(str);
```

```
constexpr int compare(size_type pos1, size_type n1, basic_string_view str,
size_type pos2, size_type n2) const;

Effects: Equivalent to: return substr(pos1, n1).compare(str.substr(pos2, n2));
```

```
constexpr int compare(const charT* s) const;

Effects: Equivalent to: return compare(basic_string_view(s));
```

```
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;

Effects: Equivalent to: return substr(pos1, n1).compare(basic_string_view(s));
```

```
constexpr int compare(size_type pos1, size_type n1, const charT* s, size_type n2) const;

Effects: Equivalent to: return substr(pos1, n1).compare(basic_string_view(s, n2));
```

```
constexpr bool starts_with(basic_string_view x) const noexcept;

Effects: Equivalent to: return substr(0, x.size()) == x;
```

```
constexpr bool starts_with(charT x) const noexcept;

Effects: Equivalent to: return !empty() && traits::eq(front(), x);
```
constexpr bool starts_with(const charT* x) const;

Effects: Equivalent to: return starts_with(basic_string_view(x));

constexpr bool ends_with(basic_string_view x) const noexcept;

Effects: Equivalent to:
\[
\text{return size()} >= \text{x.size()} \&\& \text{compare(size() - x.size(), npos, x)} == 0;
\]

constexpr bool ends_with(charT x) const noexcept;

Effects: Equivalent to:
\[
\text{return !empty()} \&\& \text{traits::eq(back(), x)};
\]

constexpr bool ends_with(const charT* x) const;

Effects: Equivalent to: return ends_with(basic_string_view(x));

21.4.2.7 Searching

Member functions in this subclause have complexity \(O(size() \cdot str.size())\) at worst, although implementations should do better.

Let \(F\) be one of find, rfind, find_first_of, find_last_of, find_first_not_of, and find_last_not_of.

(2.1) — Each member function of the form

\[
\text{constexpr return-type } F(\text{const charT}^* s, \text{size_type pos}) \text{ const} \quad \text{const};
\]

has effects equivalent to: return \(F(\text{basic_string_view}(s), \text{pos});\)

(2.2) — Each member function of the form

\[
\text{constexpr return-type } F(\text{const charT}^* s, \text{size_type pos, size_type n}) \text{ const} \quad \text{const};
\]

has effects equivalent to: return \(F(\text{basic_string_view}(s, n), \text{pos});\)

(2.3) — Each member function of the form

\[
\text{constexpr return-type } F(\text{charT} c, \text{size_type pos}) \text{ const noexcept} \quad \text{const};
\]

has effects equivalent to: return \(F(\text{basic_string_view}(\text{addressof}(c), 1), \text{pos});\)

constexpr 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:

(3.1) — pos <= xpos
(3.2) — xpos + str.size() <= size()
(3.3) — 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 \(npos\).

constexpr 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:

(6.1) — xpos <= pos
(6.2) — xpos + str.size() <= size()
(6.3) — 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 \(npos\).

constexpr 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:

(9.1) — pos <= xpos
(9.2) — xpos < size()
(9.3) — traits::eq(at(xpos), str.at(I)) for some element \(I\) of the string referenced by \(str\).
10 Effects: Determines xpos.

11 Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

```cpp
constexpr size_type find_last_of(basic_string_view str, size_type pos = npos) const noexcept;
```

12 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.

13 Effects: Determines xpos.

14 Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

```cpp
constexpr size_type find_first_not_of(basic_string_view str, size_type pos = 0) const noexcept;
```

15 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.

16 Effects: Determines xpos.

17 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;
```

18 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.

19 Effects: Determines xpos.

20 Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

### 21.4.3 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 60.

<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>
</tbody>
</table>

[Example: A sample conforming implementation for `operator==` would be:]

\[ \text{Some code here...} \]
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, type_identity_t<basic_string_view<charT, traits>> rhs) noexcept {
    return lhs.compare(rhs) == 0;
}

template<class charT, class traits>
constexpr bool operator==(type_identity_t<basic_string_view<charT, traits>> lhs, basic_string_view<charT, traits> rhs) noexcept {
    return lhs.compare(rhs) == 0;
}

Returns: 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 {
    return lhs.compare(rhs) != 0;
}

Returns: 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 {
    return lhs.compare(rhs) < 0;
}

Returns: 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 {
    return lhs.compare(rhs) > 0;
}

Returns: 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 {
    return lhs.compare(rhs) <= 0;
}

Returns: 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 {
    return lhs.compare(rhs) >= 0;
}

Returns: lhs.compare(rhs) >= 0.

21.4.4 Inserters and extractors [string.view.io]

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

21.4.5 Hash support [string.view.hash]

template<> struct hash<string_view>;

§ 21.5 754
template<> struct hash<u8string_view>;  
template<> struct hash<u16string_view>;  
template<> struct hash<u32string_view>;  
template<> struct hash<wstring_view>;  

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]

21.4.6 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 <ctctype> 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);
}  

The contents and meaning of the header <ctctype> are the same as the C standard library header <ctype.h>.

See also: ISO C 7.4

21.5.2 Header <cwctype> synopsis

namespace std {
    using wint_t = see below;  
    using wcwtrans_t = see below;  
    using wcctype_t = see below;
    
    int iswalnum(wint_t wc);  
    int iswalpha(wint_t wc);  
    int iswblank(wint_t wc);  
    int iswcntrl(wint_t wc);  
    int iswdigit(wint_t wc);  
    int iswgraph(wint_t wc);  
    int iswlower(wint_t wc);  
    int iswprint(wint_t wc);  
    int iswpunct(wint_t wc);  
    int iswupper(wint_t wc);  
    int towlower(wint_t wc);  
    int toupper(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
int iswctype(wint_t wc, wctype_t desc);
wctype_t wctype(const char* property);
wint_t towlower(wint_t wc);
wint_t toupper(wint_t wc);
wint_t towchar(wint_t wc, wctrans_t desc);
wctrans_t wctrans(const char* property);
}
#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);
    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);
    const void* memchr(const void* s, int c, size_t n);
    void* memchr(void* s, int c, size_t n);
    const char* strchr(const char* s, int c);
    char* strchr(char* s, int c);
    size_t strcspn(const char* s1, const char* s2);
    const char* strpbrk(const char* s1, const char* s2);
    char* strpbrk(char* s1, const char* s2);
    char* strrchr(const char* s, int c);
    char* strrchr(char* s, int c);
    size_t strspn(const char* s1, const char* s2);
    const char* strstr(const char* s1, const char* s2);
    char* strstr(char* s1, const char* s2);
    char* strtok(char* s1, const char* s2);
    void* memset(void* s, int c, size_t n);
    char* strerror(int errnum);
    size_t strlen(const char* s);
}
```

#define NULL see 17.2.3

1 The contents and meaning of the header `<cstring>` are the same as the C standard library header `<string.h>`.

2 The functions `strerror` and `strtok` are not required to avoid data races (16.5.5.9).

3 The functions `memcpy` and `memmove` are signal-safe (17.13.4).

4 [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

```cpp
namespace std {
    using size_t = see 17.2.4;
    using mbstate_t = see below;
    size_t strxfrm(const char* s1, const char* s2, size_t n);
    const void* memchr(const void* s, int c, size_t n);
    void* memset(void* s, int c, size_t n);
    char* strerror(int errnum);
    size_t strlen(const char* s);
}
```

§ 21.5.4 756
using wint_t = see below;

struct tm;

int fprintf(FILE* stream, const wchar_t* format, ...);
int fscanf(FILE* stream, const wchar_t* format, ...);
int wfprintf(FILE* stream, const wchar_t* format, ...);
int fwscanf(FILE* stream, const wchar_t* format, va_list arg);
int wfwscanf(FILE* stream, const wchar_t* format, va_list arg);
int vsprintf(const wchar_t* s, const wchar_t* format, va_list arg);
int vwprintf(const wchar_t* format, va_list arg);
int wprintf(const wchar_t* format, ...);
int wscanf(const wchar_t* format, ...);
wint_t fgetwc(FILE* stream);
wchar_t* fgetws(wchar_t* s, int n, FILE* stream);
wint_t fputwc(wchar_t c, FILE* stream);
int fputws(const wchar_t* s, FILE* stream);
int fwide(FILE* stream, int mode);
int fgetwc(FILE* stream);
int getwchar();
int fputwc(wchar_t c, FILE* stream);
int fputws(const wchar_t* s, FILE* stream);
int getwc(FILE* stream);
int putwc(wchar_t c, FILE* stream);
int putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE* stream);

double wcstod(const wchar_t* nptr, wchar_t** endptr);
float wcstof(const wchar_t* nptr, wchar_t** endptr);
long double wcstold(const wchar_t* nptr, wchar_t** endptr);
long int wcstol(const wchar_t* nptr, wchar_t** endptr, int base);
long long int wcstoll(const wchar_t* nptr, wchar_t** endptr, int base);
unsigned long int wcstoul(const wchar_t* nptr, wchar_t** endptr, int base);
unsigned long long int wcstoull(const wchar_t* nptr, wchar_t** endptr, int base);

wchar_t* wcscpy(wchar_t* s1, const wchar_t* s2);
wchar_t* wcsncpy(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wmemcpy(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wmemmove(wchar_t* s1, const wchar_t* s2, size_t n);
wchar_t* wcscat(wchar_t* s1, const wchar_t* s2);
wchar_t* wcsncat(wchar_t* s1, const wchar_t* s2, size_t n);
int wcscmp(const wchar_t* s1, const wchar_t* s2);
int wmemcmp(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2, size_t n);
int wcsxfrm(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s, wchar_t c); // see 16.2
wchar_t* wcscasechr(const wchar_t* s, wchar_t c); // see 16.2
size_t wcscspn(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcsncpy(const wchar_t* s1, const wchar_t* s2); // see 16.2
wchar_t* wcsprbk(const wchar_t* s1, const wchar_t* s2); // see 16.2
const wchar_t* wcsprbk(const wchar_t* s1, const wchar_t* s2); // see 16.2
wchar_t* wcschr(const wchar_t* s, wchar_t c); // see 16.2
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s, wchar_t c); // see 16.2
wchar_t* wcscat(const wchar_t* s1, const wchar_t* s2); // see 16.2
wchar_t* wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2); // see 16.2
wchar_t* wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcscat(const wchar_t* s1, const wchar_t* s2); // see 16.2
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2); // see 16.2
wchar_t* wcstok(const wchar_t* s1, const wchar_t* s2, wchar_t** ptr);
const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n); // see 16.2
size_t wcslen(const wchar_t* s);
wchar_t* wmemset(const wchar_t* s, wchar_t c, size_t n);
size_t wcftime(const wchar_t* s, size_t maxsize, const wchar_t* format, const struct tm* timeptr);
wint_t btowc(int c);
int wcscoll(const wchar_t* s1, const wchar_t* s2); // see 16.2
wchar_t* wcsnfrm(const wchar_t* s1, const wchar_t* s2, size_t n);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
const wchar_t* wcschr(const wchar_t* s1, const wchar_t* s2, size_t n);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
size_t wcsxfrm(const wchar_t* s1, const wchar_t* s2);
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`, `wcsmbbrk`, `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]

See also: ISO C 7.29

### 21.5.5 Header `<cuchar>` synopsis

```cpp
namespace std {
    using mbstate_t = see below;
    using size_t = see 17.2.4;

    size_t mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);
    size_t c8rtomb(char* s, char8_t c8, mbstate_t* ps);
    size_t mbrtoc16(char16_t* pc16, const char* s, size_t n, mbstate_t* ps);
    size_t c16rtomb(char* s, char16_t c16, mbstate_t* ps);
    size_t mbrtoc32(char32_t* pc32, const char* s, size_t n, mbstate_t* ps);
    size_t c32rtomb(char* s, char32_t c32, mbstate_t* ps);
}
```

The contents and meaning of the header `<cuchar>` are the same as the C standard library header `<uchar.h>`, except that it declares the additional `mbrtoc8` and `c8rtomb` functions and does not declare types `char16_t` nor `char32_t`.

See also: ISO C 7.28

### 21.5.6 Multibyte / wide string and character conversion functions

[Note: The headers `<cstdlib>` (17.2.2), `<cuchar>` (21.5.5), and `<cwchar>` (21.5.4) declare the functions described in this subclause. —end note]

```cpp
int mbsinit(const mbstate_t* ps);
int mblen(const char* s, size_t n);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);
```

Effects: These functions have the semantics specified in the C standard library.

See also: ISO C 7.22.7.1, 7.22.8, 7.29.6.2.1

```cpp
int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);
```

Effects: These functions have the semantics specified in the C standard library.

Remarks: Calls to these functions may introduce a data race (16.5.5.9) with other calls to the same function.

See also: ISO C 7.22.7
size_t wcsrtombs(char* dst, const wchar_t** src, size_t len, mbstate_t* ps);

Effects: These functions have the semantics specified in the C standard library.

Remarks: Calling these functions with an mbstate_t* argument that is a null pointer value may introduce a data race (16.5.5.9) with other calls to the same function with an mbstate_t* argument that is a null pointer value.

See also: ISO C 7.29.6.3

size_t mbrtoc8(char8_t* pc8, const char* s, size_t n, mbstate_t* ps);

Effects: If s is a null pointer, equivalent to mbrtoc8(nullptr, "", 1, ps). Otherwise, the function inspects at most n bytes beginning with the byte pointed to by s to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding UTF-8 code units and then, if pc8 is not a null pointer, stores the value of the first (or only) such code unit in the object pointed to by pc8. Subsequent calls will store successive UTF-8 code units without consuming any additional input until all the code units have been stored. If the corresponding Unicode character is U+0000, the resulting state described is the initial conversion state.

Returns: The first of the following that applies (given the current conversion state):

(8.1) 0, if the next n or fewer bytes complete the multibyte character that corresponds to the U+0000 Unicode character (which is the value stored).

(8.2) between 1 and n (inclusive), if the next n or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

(8.3) (size_t)(-3), if the next character resulting from a previous call has been stored (no bytes from the input have been consumed by this call).

(8.4) (size_t)(-2), if the next n bytes contribute to an incomplete (but potentially valid) multibyte character, and all n bytes have been processed (no value is stored).

(8.5) (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 (16.5.5.9) 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 61.

### Table 61 — Containers library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.2 Requirements</td>
<td>&lt;array&gt;</td>
</tr>
<tr>
<td>22.3 Sequence containers</td>
<td>&lt;deque&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;forward_list&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;list&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;vector&gt;</td>
</tr>
<tr>
<td>22.4 Associative containers</td>
<td>&lt;map&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;set&gt;</td>
</tr>
<tr>
<td>22.5 Unordered associative containers</td>
<td>&lt;unordered_map&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;unordered_set&gt;</td>
</tr>
<tr>
<td>22.6 Container adaptors</td>
<td>&lt;queue&gt;</td>
</tr>
<tr>
<td></td>
<td>&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 62, 63, and 64 X denotes a container class containing objects of type T, a and b denote values of type X, u denotes an identifier, r denotes a non-const value of type X, and rv denotes a non-const rvalue of type X.

### Table 62 — 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>Expects: 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>
</tbody>
</table>
Table 62 — Container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note Complexity</th>
</tr>
</thead>
</table>
| X::const-
  reference | const T& | semantie pre-/post-condition | compile time |
| X::iterator | iterator type whose value type is T | any iterator category that meets the forward iterator requirements. convertible to X::const_iterator. | compile time |
| X::const-
  iterator | constant iterator type whose value type is T | any iterator category that meets the forward iterator requirements. | compile time |
| X::difference_type | signed integer type | is identical to the difference type of X::iterator and X::const_iterator | compile time |
| X::size_type | unsigned integer type | size_type can represent any non-negative value of difference_type | compile time |
| X u; | Ensures: u.empty() | constant |
| X() | Ensures: X().empty() | constant |
| X(a) | Expects: T is Cpp17CopyInsertable into X (see below). Ensures: a == X(a). | linear |
| X u(a); X u = a; | Expects: T is Cpp17CopyInsertable into X (see below). Ensures: u == a | linear |
| X u(rv); X u = rv; | Ensures: u is equal to the value that rv had before this construction | (Note B) |
| a = rv X& | All existing elements of a are either move assigned to or destroyed Ensures: a is equal to the value that rv had before this assignment | linear |
| a.-X() | void | Effects: destroys every element of a; any memory obtained is deallocated. | linear |
| a.begin() | iterator; const_iterator for constant a | constant |
| a.end() | iterator; const_iterator for constant a | constant |
| a.cbegin() | const_iterator const_cast<X const&>(a).begin(); | constant |
Table 62 — Container requirements (continued)

<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>a.cend()</td>
<td>const_iterator</td>
<td>const_cast&lt;X const&amp;(a).end&gt;();</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation. equal(a.begin(), a.end(), b.begin(), b.end());</td>
<td>Expects: T meets the Cpp17EqualityComparable requirements</td>
<td>Constant if a.size() != b.size(), linear otherwise</td>
</tr>
<tr>
<td>a != b</td>
<td>convertible to bool</td>
<td>Equivalent to !(a == b)</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td>Equivalent to a.swap(b)</td>
<td></td>
<td>(Note A)</td>
</tr>
<tr>
<td>r = a</td>
<td>X&amp;</td>
<td>Ensures: r == a.</td>
<td></td>
<td>linear</td>
</tr>
<tr>
<td>a.size()</td>
<td>size_type</td>
<td>distance(a.begin(), a.end())</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.max_size()</td>
<td>size_type</td>
<td>distance(begin(), end()) for the largest possible container</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.empty()</td>
<td>convertible to bool</td>
<td>a.begin() == a.end()</td>
<td></td>
<td>constant</td>
</tr>
</tbody>
</table>

Those entries marked “(Note A)” or “(Note B)” have linear complexity for array and have constant complexity for all other standard containers. [Note: 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

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<unspecified>, 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 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 satisfies the additional requirements in Table 63.

Table 63 — Reversible container requirements

<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::reverse_iterator</td>
<td>iterator type whose value type is T</td>
<td>reverse_iterator&lt;iterator&gt;</td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_reverse_iterator</td>
<td>constant iterator type whose value type is T</td>
<td>reverse_iterator&lt;const_reverse_iterator&gt;</td>
<td>compile time</td>
</tr>
<tr>
<td>a.rbegin()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(end())</td>
<td>constant</td>
</tr>
<tr>
<td>a.rend()</td>
<td>reverse_iterator; const_reverse_iterator for constant a</td>
<td>reverse_iterator(begin())</td>
<td>constant</td>
</tr>
<tr>
<td>a.crbegin()</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rbegin()</td>
<td>constant</td>
</tr>
<tr>
<td>a.crend()</td>
<td>const_reverse_iterator</td>
<td>const_cast&lt;X const&amp;&gt;(a).rend()</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.
(11.6) — 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]
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.

A contiguous container is a container whose member types iterator and const_iterator meet the Cpp17RandomAccessIterator requirements (23.3.5.6) and model ContiguousIterator (23.3.4.14).

Table 64 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 64 unless otherwise stated. If the iterators passed to lexicographical_compare satisfy the constexpr iterator requirements (23.3.1) then the operations described in Table 64 are implemented by constexpr functions.

Table 64 — Optional container operations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>convertible to bool</td>
<td>lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())</td>
<td>Expects: <code>&lt;</code> is defined for values of type (possibly const) T; <code>&lt;</code> is a total ordering relationship.</td>
<td>linear</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>convertible to bool</td>
<td>b &lt; a</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>convertible to bool</td>
<td>!(a &gt; b)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>convertible to bool</td>
<td>!(a &lt; b)</td>
<td>linear</td>
<td></td>
</tr>
</tbody>
</table>

[Note: The algorithm lexicographical_compare() is defined in Clause 25. — end note]

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 65.

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 from args, for zero or more arguments args, means that the following expression is well-formed:

```
allocator_traits<A>::construct(m, p, args)
```

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)
```
© ISO/IEC 841.9

— \( T \) is *C++17Erasable from \( X \) means that the following expression is well-formed:

\[
\text{allocator_traits}<A>::\text{destroy}(m, p)
\]

[Note: A container calls \( \text{allocator_traits}<A>::\text{construct}(m, p, \text{args}) \) to construct an element at \( p \) using \( \text{args} \), with \( m == \text{get_allocator()} \). The default \text{construct} in \text{allocator} will call \( ::\text{new}((\text{void}*)p) \) \( T(\text{args}) \), but specialized allocators may choose a different definition. — end note]

16

In Table 65, \( X \) denotes an allocator-aware container class with a \text{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 65 — Allocator-aware container requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocator_type A</td>
<td></td>
<td><strong>Mandates:</strong> allocator_type::value_type is the same as ( X::\text{value_type} ).</td>
<td>compile time</td>
</tr>
<tr>
<td>get_allocator()</td>
<td>A</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>( X() ) ( X u; )</td>
<td></td>
<td><strong>Expects:</strong> ( A ) meets the C++17DefaultConstructible requirements. <strong>Ensures:</strong> ( u.\text{empty}() ) returns true, ( u.\text{get_allocator}() == A() )</td>
<td>constant</td>
</tr>
<tr>
<td>( X(m) ) ( X u(m); )</td>
<td></td>
<td><strong>Ensures:</strong> ( u.\text{empty}() ) returns true, ( u.\text{get_allocator}() == m )</td>
<td>constant</td>
</tr>
<tr>
<td>( X(t, m) ) ( X u(t, m); )</td>
<td></td>
<td><strong>Expects:</strong> ( T ) is C++17CopyInsertable into ( X ). <strong>Ensures:</strong> ( u == t ), ( u.\text{get_allocator}() == m )</td>
<td>linear</td>
</tr>
<tr>
<td>( X(rv) ) ( X u(rv); )</td>
<td></td>
<td><strong>Ensures:</strong> ( u ) has the same elements as ( rv ) had before this construction; the value of ( u.\text{get_allocator}() ) is the same as the value of ( rv.\text{get_allocator}() ) before this construction.</td>
<td>constant</td>
</tr>
<tr>
<td>( X(rv, m) ) ( X u(rv, m); )</td>
<td></td>
<td><strong>Expects:</strong> ( T ) is C++17MoveInsertable into ( X ). <strong>Ensures:</strong> ( u ) has the same elements, or copies of the elements, that ( rv ) had before this construction, ( u.\text{get_allocator}() == m )</td>
<td>constant if ( m == rv.\text{get_allocator()} ), otherwise linear</td>
</tr>
<tr>
<td>( a = t )</td>
<td>( X&amp; )</td>
<td><strong>Expects:</strong> ( T ) is C++17CopyInsertable into ( X ) and C++17CopyAssignable. <strong>Ensures:</strong> ( a == t )</td>
<td>linear</td>
</tr>
</tbody>
</table>
### Table 65 — Allocator-aware container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = rv X&amp;</td>
<td>X&amp;</td>
<td><em>Expects:</em> If <code>allocator_traits&lt;allocator_type&gt;::propagate_on_container_move_assignment::value</code> is false, <code>T</code> is <code>Cpp17MoveInsertable</code> into <code>X</code> and <code>Cpp17MoveAssignable</code>. <em>Effects:</em> All existing elements of <code>a</code> are either move assigned to or destroyed. <em>Ensures:</em> <code>a</code> is equal to the value that <code>rv</code> had before this assignment.</td>
<td>linear</td>
</tr>
<tr>
<td>a.swap(b)</td>
<td>void</td>
<td><em>Effects:</em> exchanges the contents of <code>a</code> and <code>b</code></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 satisfies both of the following conditions:

- The qualified-id `A::value_type` is valid and denotes a type (13.9.2).
- The expression `declval<A&>().allocate(size_t{})` is well-formed when treated as an unevaluated operand.

22.2.2 Container data races

1 For purposes of avoiding data races (16.5.5.9), 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 16.5.5.9, 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

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]*
3 In Tables 66 and 67, X denotes a sequence container class, a denotes a value of type X containing elements of type T, u denotes the name of a variable being declared, A denotes X::allocator_type if the qualified-id X::allocator_type is valid and denotes a type (13.9.2) and allocator<T> if it doesn’t, i and j denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to value_type, [i, j) denotes a valid range, il designates an object of type initializer_list<value_type>, n denotes a value of type X::size_type, p denotes a valid constant iterator to a, q denotes a valid dereferenceable constant iterator to a, [q1, q2) denotes a valid range of constant iterators in a, t denotes an lvalue or a const rvalue of X::value_type, and rv denotes a non-const rvalue of X::value_type. Args denotes a template parameter pack; args denotes a function parameter pack with the pattern Args&&.

4 The complexities of the expressions are sequence dependent.

Table 66 — Sequence container requirements (in addition to container)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(n, t)</td>
<td></td>
<td><strong>Expects:</strong> T is Cpp17CopyInsertable into X.</td>
</tr>
<tr>
<td>X u(n, t);</td>
<td></td>
<td><strong>Ensures:</strong> distance(begin(), end()) == n.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Constructs a sequence container with n copies of t.</td>
</tr>
<tr>
<td>X(i, j)</td>
<td></td>
<td><strong>Expects:</strong> T is Cpp17EmplaceConstructible into X from *i. For vector, if the iterator does not meet the Cpp17ForwardIterator requirements (23.3.5.4), T is also Cpp17MoveInsertable into X.</td>
</tr>
<tr>
<td>X u(i, j);</td>
<td></td>
<td><strong>Ensures:</strong> distance(begin(), end()) == distance(i, j)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Constructs a sequence container equal to the range [i, j). Each iterator in the range [i, j) is dereferenced exactly once.</td>
</tr>
<tr>
<td>X(il)</td>
<td></td>
<td>Equivalent to X(il.begin(), il.end())</td>
</tr>
<tr>
<td>a = il</td>
<td>X&amp;</td>
<td><strong>Expects:</strong> T is Cpp17CopyInsertable into X and Cpp17CopyAssignable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Assigns the range [il.begin(), il.end()] into a. All existing elements of a are either assigned to or destroyed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Returns:</strong> *this.</td>
</tr>
<tr>
<td>aemplace(p, args)</td>
<td>iterator</td>
<td><strong>Expects:</strong> T is Cpp17EmplaceConstructible into X from args. For vector and deque, T is also Cpp17MoveInsertable into X and Cpp17MoveAssignable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Inserts an object of type T constructed with std::forward&lt;Args&gt;(args)... before p. [Note: args may directly or indirectly refer to a value in a. — end note]</td>
</tr>
<tr>
<td>a.insert(p, t)</td>
<td>iterator</td>
<td><strong>Expects:</strong> T is Cpp17CopyInsertable into X. For vector and deque, T is also Cpp17CopyAssignable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Inserts a copy of t before p.</td>
</tr>
<tr>
<td>a.insert(p, rv)</td>
<td>iterator</td>
<td><strong>Expects:</strong> T is Cpp17MoveInsertable into X. For vector and deque, T is also Cpp17MoveAssignable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Inserts a copy of rv before p.</td>
</tr>
<tr>
<td>a.insert(p, n, t)</td>
<td>iterator</td>
<td><strong>Expects:</strong> T is Cpp17CopyInsertable into X and Cpp17CopyAssignable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Effects:</strong> Inserts n copies of t before p.</td>
</tr>
</tbody>
</table>
Table 66 — Sequence container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, i, j)</code></td>
<td>iterator</td>
<td><code>Expects: 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.</code></td>
</tr>
<tr>
<td><code>a.insert(p, il)</code></td>
<td>iterator</td>
<td><code>a.insert(p, il.begin(), il.end()).</code></td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td>iterator</td>
<td><code>Expects: For vector and deque, T is Cpp17MoveAssignable. Effects: Erases the element pointed to by q.</code></td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td>iterator</td>
<td><code>Expects: For vector and deque, T is Cpp17MoveAssignable. Effects: Erases the elements in the range [q1, q2).</code></td>
</tr>
<tr>
<td><code>a.clear()</code></td>
<td>void</td>
<td><code>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. Ensures: a.empty() is true. Complexity: Linear.</code></td>
</tr>
<tr>
<td><code>a.assign(i, j)</code></td>
<td>void</td>
<td><code>Expects: 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.</code></td>
</tr>
<tr>
<td><code>a.assign(il)</code></td>
<td>void</td>
<td><code>a.assign(il.begin(), il.end()).</code></td>
</tr>
<tr>
<td><code>a.assign(n, t)</code></td>
<td>void</td>
<td><code>Expects: 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.</code></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.

7 The iterator returned from `a.insert(p, n, t)` points to the copy of the first element inserted into a, or p if n == 0.
The iterator returned from `a.insert(p, i, j)` points to the copy of the first element inserted into `a`, or `p` if `i == j`.

The iterator returned from `a.insert(p, il)` points to the copy of the first element inserted into `a`, or `p` if `il` is empty.

The iterator returned from `a.emplace(p, args)` points to the new element constructed from `args` into `a`.

The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

The iterator returned by `a.erase(q1, q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

For every sequence container defined in this Clause and in Clause 21:

(13.1) — If the constructor

```cpp
template<class InputIterator>
X(InputIterator first, InputIterator last,
   const allocator_type& alloc = allocator_type());
```

is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor shall not participate in overload resolution.

(13.2) — If the member functions of the forms:

```cpp
template<class InputIterator>
return-type F(const_iterator p,
               InputIterator first, InputIterator last); // such as insert
```

```cpp
template<class InputIterator>
return-type F(InputIterator first, InputIterator last); // such as append, assign
```

```cpp
template<class InputIterator>
return-type F(const_iterator i1, const_iterator i2,
               InputIterator first, InputIterator last); // such as replace
```

are called with a type `InputIterator` that does not qualify as an input iterator, then these functions shall not participate in overload resolution.

(13.3) — A deduction guide for a sequence container shall not participate in overload resolution if it has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter, or if it has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.

Table 67 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the “container” column, and shall implement them so as to take amortized constant time.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.front()</code></td>
<td>reference; const_reference</td>
<td>*a.begin()</td>
<td>basic_string, array, deque, forward_list, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for constant a</td>
<td></td>
</tr>
<tr>
<td><code>a.back()</code></td>
<td>reference; const_reference</td>
<td>{ auto tmp = a.end(); --tmp; return *tmp; }</td>
<td>basic_string, array, deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for constant a</td>
<td></td>
</tr>
<tr>
<td><code>a.emplace_front(args)</code></td>
<td>reference</td>
<td>Effects: Prepends an object of type T constructed with std::forward&lt;Args&gt;(args).... Expects: T is Cpp17EmplaceConstructible into X from args. Returns: a.front().</td>
<td>deque, forward_list, list</td>
</tr>
</tbody>
</table>
Table 67 — 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><code>a.emplace_back(args)</code></td>
<td>reference</td>
<td>Effects: Appends an object of type <code>T</code> constructed with <code>std::forward&lt;Args&gt;(args)....</code></td>
<td>deque, list, vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Expects: <code>T</code> is <code>Cpp17EmplaceConstructible</code> into <code>X</code> from <code>args</code>. For <code>vector</code>, <code>T</code> is also <code>Cpp17MoveInsertable</code> into <code>X</code>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns: <code>a.back()</code></td>
<td></td>
</tr>
<tr>
<td><code>a.push_front(t)</code></td>
<td>void</td>
<td>Effects: Prepends a copy of <code>t</code>. Expects: <code>T</code> is <code>Cpp17CopyInsertable</code> into <code>X</code>.</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td><code>a.push_front(rv)</code></td>
<td>void</td>
<td>Effects: Prepends a copy of <code>rv</code>. Expects: <code>T</code> is <code>Cpp17MoveInsertable</code> into <code>X</code>.</td>
<td>basic_string, vector</td>
</tr>
<tr>
<td><code>a.push_back(t)</code></td>
<td>void</td>
<td>Effects: Appends a copy of <code>t</code>. Expects: <code>T</code> is <code>Cpp17CopyInsertable</code> into <code>X</code>.</td>
<td>basic_string, vector</td>
</tr>
<tr>
<td><code>a.push_back(rv)</code></td>
<td>void</td>
<td>Effects: Appends a copy of <code>rv</code>. Expects: <code>T</code> is <code>Cpp17MoveInsertable</code> into <code>X</code>.</td>
<td>basic_string, vector</td>
</tr>
<tr>
<td><code>a.pop_front()</code></td>
<td>void</td>
<td>Effects: Destroys the first element. Expects: <code>a.empty()</code> is <code>false</code>.</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td><code>a.pop_back()</code></td>
<td>void</td>
<td>Effects: Destroys the last element. Expects: <code>a.empty()</code> is <code>false</code>.</td>
<td>basic_string, vector</td>
</tr>
<tr>
<td><code>a[n]</code></td>
<td>reference; const_reference</td>
<td>*(a.begin() + n)</td>
<td>basic_string, array, vector</td>
</tr>
<tr>
<td></td>
<td>for constant <code>a</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a.at(n)</code></td>
<td>reference; const_reference</td>
<td>*(a.begin() + n)</td>
<td>basic_string, array, vector</td>
</tr>
<tr>
<td></td>
<td>for constant <code>a</code></td>
<td></td>
<td></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 68.

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.
Table 68 — Container types with compatible nodes

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Equivalent Container Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>map&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>multimap&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>set&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>multiset&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, H1, E1, A&gt;</td>
<td>unordered_map&lt;K, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, H1, E1, A&gt;</td>
<td>unordered_multimap&lt;K, 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>

template<unspecified>

class node-handle {
public:
   // These type declarations are described in Tables 69 and 70.
   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
   node-handle(node-handle&& nh) noexcept : ptr_(nh.ptr_), alloc_(nh.alloc_)
   {
   }

   node-handle() {} // 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(x.swap(y))
      { x.swap(y);
      }
};

22.2.4.2 Constructors, copy, and assignment

node-handle(node-handle&& nh) noexcept;

1 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);  

2  
Expects: Either !alloc_ or ator_traits::propagate_on_container_move_assignment::value is true, or alloc_ == nh.alloc_.  

3  
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_.  

4  
Returns: *this.  

5  
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;  

1  
Expects: empty() == false.  

2  
Returns: A reference to the value_type subobject in the container_node_type object pointed to by ptr_.  

3  
Throws: Nothing.

key_type& key() const;  

4  
Expects: empty() == false.  

5  
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_.  

6  
Throws: Nothing.  

7  
Remarks: Modifying the key through the returned reference is permitted.

mapped_type& mapped() const;  

8  
Expects: empty() == false.  

9  
Returns: A reference to the mapped_type member of the value_type subobject in the container_node_type object pointed to by ptr_.  

10  
Throws: Nothing.

allocator_type get_allocator() const;  

11  
Expects: empty() == false.  

12  
Returns: *alloc_.  

13  
Throws: Nothing.  

explicit operator bool() const noexcept;  

14  
Returns: ptr_ != nullptr.  

[[nodiscard]] bool empty() const noexcept;  

15  
Returns: ptr_ == nullptr.
In Table 69, 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 68), 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, \texttt{a\_tran} denotes a possibly \texttt{const} value of type $X$ when the \texttt{qualified-id} $X::\texttt{key\_compare}\::\texttt{is\_transparent}$ is valid and denotes a type (13.9.2), \texttt{i} and \texttt{j} satisfy input iterator requirements and refer to elements implicitly convertible to \texttt{value\_type}, $[i, j)$ denotes a valid range, \texttt{p} denotes a valid constant iterator to \texttt{a}, \texttt{q} denotes a valid dereferenceable constant iterator to \texttt{a}, \texttt{r} denotes a valid dereferenceable iterator to \texttt{a}, $[q_1, q_2)$ denotes a valid range of constant iterators in \texttt{a}, \texttt{il} designates an object of type \texttt{initializer\_list<value\_type>}, \texttt{t} denotes a value of type $X::\texttt{value\_type}$, \texttt{k} denotes a value of type $X::\texttt{key\_type}$ and \texttt{c} denotes a possibly \texttt{const} value of type $X::\texttt{key\_compare}$; $kl$ is a value such that \texttt{a} is partitioned (25.7) with respect to $c(r, kl)$, with $r$ the key value of $e$ and $e$ in \texttt{a}; $ku$ is a value such that \texttt{a} is partitioned with respect to $!c(ku, r)$; $ke$ is a value such that \texttt{a} is partitioned with respect to $c(r, ke)$ and $!c(ke, r)$, with $c(r, ke)$ implying $!c(ke, r)$. \texttt{A} denotes the storage allocator used by \texttt{X}, if any, or \texttt{allocator<X::value\_type>} otherwise, \texttt{m} denotes an allocator of a type convertible to \texttt{A}, and \texttt{nh} denotes a non-const rvalue of type $X::\texttt{node\_type}$.

Table 69 — Associative container requirements (in addition to container)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X::\texttt{key_type}$</td>
<td>$\texttt{Key}$</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\texttt{mapped_}-\texttt{type}$ (map and multimap only)</td>
<td>T</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\texttt{value_}-\texttt{type}$ (set and multiset only)</td>
<td>$\texttt{Key}$</td>
<td>$\texttt{Expects: value_type}$ is \texttt{Ccpp17Erasable} from $X$</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\texttt{value_}-\texttt{type}$ (map and multimap only)</td>
<td>pair&lt;const $\texttt{Key}$, T&gt;</td>
<td>$\texttt{Expects: value_type}$ is \texttt{Ccpp17Erasable} from $X$</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\texttt{key_}-\texttt{compare}$</td>
<td>Compare</td>
<td>$\texttt{Expects: key_compare}$ is \texttt{Ccpp17CopyConstructible}</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\texttt{value_}-\texttt{compare}$</td>
<td>a binary predicate type</td>
<td>is the same as $\texttt{key_compare}$ for set and multiset; is an ordering relation on pairs induced by the first component (i.e., Key) for map and multimap.</td>
<td>compile time</td>
</tr>
<tr>
<td>$X::\texttt{node_}-\texttt{type}$</td>
<td>a specialization of a \texttt{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>$X(c)$</td>
<td>$X \ u(c)$;</td>
<td>$\texttt{Effects:}$ Constructs an empty constant container. Uses a copy of $c$ as a comparison object.</td>
<td></td>
</tr>
</tbody>
</table>
Table 69 — 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>expecting <code>key_compare</code> meets the <code>Cpp17DefaultConstructible</code> requirements. Effects: Constructs an empty container. Uses <code>Compare()</code> as a comparison object</td>
<td>constant</td>
</tr>
<tr>
<td>X(i,j,c)</td>
<td>X u(i,j,c);</td>
<td>expecting <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> 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 <code>distance(i, j); linear if [i, j) is sorted with </code>value_comp()`</td>
</tr>
<tr>
<td>X(i,j)</td>
<td>X u(i,j);</td>
<td>expecting <code>key_compare</code> meets the <code>Cpp17DefaultConstructible</code> requirements. <code>value_type</code> is <code>Cpp17EmplaceConstructible</code> into X from *i. Effects: Same as above, but uses <code>Compare()</code> 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 X(il.begin(), il.end())</td>
<td>same as above</td>
</tr>
<tr>
<td>X(il,c)</td>
<td>same as X(il.begin(), il.end(), c)</td>
<td>same as X(il.begin(), il.end(), c)</td>
<td>same as above</td>
</tr>
<tr>
<td>a = il</td>
<td>Xk</td>
<td>expecting <code>value_type</code> is <code>Cpp17CopyInsertable</code> into X and <code>Cpp17CopyAssignable</code>. 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 <code>il.size() + a.size(); linear if [il.begin(), il.end()) is sorted with </code>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 <code>value_compare</code> constructed out of the comparison object</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 69 — 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></td>
<td></td>
<td>Expects: value_type is Cpp17EmplaceConstructible into X from args. Effects: Inserts a value_type object t constructed with 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.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_uniq. emplace(args)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td></td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_eq. emplace(args)</td>
<td>iterator</td>
<td>Expects: value_type is Cpp17EmplaceConstructible into X from args. Effects: Inserts a value_type object t constructed with 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.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a.emplace_hint(p, args)</td>
<td>iterator</td>
<td>equivalent to a.emplace(std::forward&lt;Args&gt;(args)...). Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The element is inserted as close as possible to the position just prior to p.</td>
<td>logarithmic in general, but amortized constant if the element is inserted right before p</td>
</tr>
<tr>
<td>a_uniq. insert(t)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Expects: If t is a non-const value, value_type is Cpp17MoveInsertable into X; otherwise, value_type is Cpp17CopyInsertable into X. Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 69 — 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_eq.insert(t)</td>
<td>iterator</td>
<td>Expects: If ( t ) is a non-const rvalue, ( \text{value_type} ) is \text{Cpp17MoveInsertable} into ( X ); otherwise, ( \text{value_type} ) 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 ( \text{a_eq} ), ( t ) is inserted at the end of that range.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a.insert(p, t)</td>
<td>iterator</td>
<td>Expects: If ( t ) is a non-const rvalue, ( \text{value_type} ) is \text{Cpp17MoveInsertable} into ( X ); otherwise, ( \text{value_type} ) 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(i, j)</td>
<td>void</td>
<td>Expects: %\text{value_type} is \text{Cpp17EmplaceConstructible} into ( X ) from (*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.\text{size()} + N) ), where ( N ) has the value ( \text{distance}(i, j) )</td>
</tr>
<tr>
<td>a.insert(il)</td>
<td>void</td>
<td>equivalent to ( \text{a.insert(il.begin(), il.end())} )</td>
<td></td>
</tr>
</tbody>
</table>
Table 69 — Associative container requirements (in addition to
container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_uniq.</td>
<td>insert_-</td>
<td>Expects: 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(). Ensures: If nh is empty, inserted is false, position is end(), and node is empty. Otherwise if the insertion took place, inserted is true, position points to the inserted element, and node is empty; if the insertion failed, inserted is false, node has the previous value of nh, and position points to an element with a key equivalent to nh.key().</td>
<td>logarithmic</td>
</tr>
<tr>
<td>insert(nh)</td>
<td>iterator</td>
<td>Expects: 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. Ensures: nh is empty.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_eq.</td>
<td>iterator</td>
<td>Expects: 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. Ensures: nh is empty.</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 69 — 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.insert(p, nh)</td>
<td>iterator</td>
<td>Expects: nh is empty or a.get_allocator() == nh.get_allocator(). Effects: 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 element is inserted as close as possible to the position just prior to p. Ensures: nh is empty if insertion succeeds, unchanged if insertion fails.</td>
<td>logarithmic in general, but amortized constant if the element is inserted right before p.</td>
</tr>
<tr>
<td>a.extract(k)</td>
<td>node_type</td>
<td>Effects: Removes the first element in the container with key equivalent to k. Returns: A node_type owning the element if found, otherwise an empty node_type.</td>
<td>log(a.size())</td>
</tr>
<tr>
<td>a.extract(q)</td>
<td>node_type</td>
<td>Effects: Removes the element pointed to by q. Returns: A node_type owning that element.</td>
<td>amortized constant</td>
</tr>
</tbody>
</table>
### Table 69 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.merge(a2)</code></td>
<td>void</td>
<td><em>Expects:</em> <code>a.get_allocator()</code> == <code>a2.get_allocator()</code>.&lt;br&gt; <em>Effects:</em> 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>. <em>Ensures:</em> 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>. <em>Throws:</em> Nothing unless the comparison object throws.</td>
<td>$N \log(a.size()+N)$, where $N$ has the value <code>a2.size()</code></td>
</tr>
<tr>
<td><code>a.erase(k)</code></td>
<td>size_type</td>
<td><em>Effects:</em> Erases all elements in the container with key equivalent to <code>k</code>. <em>Returns:</em> The number of erased elements.</td>
<td>$\log(a.size()) + a.count(k)$</td>
</tr>
<tr>
<td><code>a.erase(q)</code></td>
<td>iterator</td>
<td><em>Effects:</em> Erases the element pointed to by <code>q</code>. <em>Returns:</em> An iterator pointing to the element immediately following <code>q</code> prior to the element being erased. If no such element exists, returns <code>a.end()</code>.</td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>a.erase(r)</code></td>
<td>iterator</td>
<td><em>Effects:</em> Erases the element pointed to by <code>r</code>. <em>Returns:</em> An iterator pointing to the element immediately following <code>r</code> prior to the element being erased. If no such element exists, returns <code>a.end()</code></td>
<td>amortized constant</td>
</tr>
<tr>
<td><code>a.erase(q1, q2)</code></td>
<td>iterator</td>
<td><em>Effects:</em> Erases all the elements in the range <code>[q1, q2)</code>. <em>Returns:</em> An iterator pointing to the element pointed to by <code>q2</code> prior to any elements being erased. If no such element exists, <code>a.end()</code> is returned.</td>
<td>$\log(a.size()) + N$, where $N$ has the value <code>distance(q1, q2)</code></td>
</tr>
</tbody>
</table>
Table 69 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Effects: Equivalent to a.erase(a.begin(), a.end()). Ensures: a.empty() is true.</td>
<td>linear in a.size().</td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator; const_iterator for constant b.</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>a_tran.find(ke)</td>
<td>iterator; const_iterator for constant a_tran.</td>
<td>Returns: An iterator pointing to an element with key r such that !c(r, ke) &amp;&amp; !c(ke, r), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>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 !c(r, ke) &amp;&amp; !c(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>
<tr>
<td>b.lower_bound(k)</td>
<td>iterator; const_iterator for constant b.</td>
<td>Returns: An iterator pointing to the first element with key not less than k, or b.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran.lower_bound(kl)</td>
<td>iterator; const_iterator for constant a_tran.</td>
<td>Returns: An iterator pointing to the first element with key r such that !c(r, kl), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>b.upper_bound(k)</td>
<td>iterator; const_iterator for constant b.</td>
<td>Returns: An iterator pointing to the first element with key greater than k, or b.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>a_tran.upper_bound(ku)</td>
<td>iterator; const_iterator for constant a_tran.</td>
<td>Returns: An iterator pointing to the first element with key r such that c(ku, r), or a_tran.end() if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td>b.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;: pair&lt;const_iterator, const_iterator&gt; for constant b.</td>
<td>Effects: Equivalent to: return make_pair(b.lower_bound(k), b.upper_bound(k));</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 69 — 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_tran.</td>
<td>pair&lt;</td>
<td>Effects: Equivalent to: return</td>
<td>logarithmic</td>
</tr>
<tr>
<td>equal_-</td>
<td>iterator,</td>
<td>make_pair(</td>
<td></td>
</tr>
<tr>
<td>range(ke)</td>
<td>iterator&gt;;</td>
<td>a_tran.lower_bound(ke),</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pair&lt;const_-</td>
<td>a_tran.upper_bound(ke));</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iterator,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>const_-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>iterator&gt; for</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_tran.</td>
<td></td>
<td></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:

\[
\text{value\_comp}(*j, *i) == \text{false}
\]

12 For associative containers with unique keys the stronger condition holds:

\[
\text{value\_comp}(*i, *j)! = \text{false}
\]

13 When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, through either a copy constructor or an assignment operator, the target container shall then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

14 The member function templates find, count, contains, lower_bound, upper_bound, and equal_range shall not participate in overload resolution unless the qualified-id Compare::is\_transparent is valid and denotes a type (13.9.2).

15 A deduction guide for an associative container shall not participate in overload resolution if any of the following are true:

(15.1) — It has an InputIterator template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(15.2) — It has an Allocator template parameter and a type that does not qualify as an allocator is deduced for that parameter.

(15.3) — It has a Compare template parameter and a type that qualifies as an allocator is deduced for that parameter.

22.2.6.1 Exception safety guarantees [associative.reqmts.except]

1 For associative containers, no clear() function throws an exception. erase(k) does not throw an exception unless that exception is thrown by the container’s Compare object (if any).

2 For associative containers, if an exception is thrown by any operation from within an insert or emplace function inserting a single element, the insertion has no effect.

3 For associative containers, no swap function throws an exception unless that exception is thrown by the swap of the container’s Compare object (if any).
22.2.7 Unordered associative containers

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 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]

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 65 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 70:

- 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 68),
- 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-id X::hasher::transparent_key_equal is valid and denotes a type (13.9.2),

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,

hf denotes a possibly const value of type hasher,
eq denotes a possibly const value of type key_equal,

ke is a value such that

eq(r1, ke) == eq(ke, r1)

hf(r1) == hf(ke) if eq(r1, ke) is true, and

(eq(r1, ke) && eq(r1, r2)) == eq(r2, ke)

where r1 and r2 are keys of elements in a_tran,

n denotes a value of type size_type,
z denotes a value of type float,

nh denotes a non-const rvalue of type X::node_type, and

hk and hke denote values of type size_t.

Table 70 — Unordered associative container requirements (in addition to container)

<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 (unordered_map and unordered_multimap only)</td>
<td>T</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::value_type (unordered_set and unordered_multiset only)</td>
<td>Key</td>
<td>Expects: 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>Expects: value_type is Cpp17Erasable from X</td>
<td>compile time</td>
</tr>
<tr>
<td>X::hasher</td>
<td>Hash</td>
<td>Expects: 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>Hash::transparent_key_equal if such a qualified-id is valid and denotes a type (13.9.2); otherwise, Pred.</td>
<td>Expects: 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>
</tbody>
</table>
Table 70 — 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><code>X::local_iterator</code></td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as <code>X::iterator</code>'s.</td>
<td>A <code>local_iterator</code> 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><code>X::const_local_iterator</code></td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as <code>X::const_iterator</code>'s.</td>
<td>A <code>const_local_iterator</code> 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><code>X::node_type</code></td>
<td>A specialization of a <code>node-handle</code> 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>
<tr>
<td><code>X(n, hf, eq)</code></td>
<td><code>X</code> <code>a(n, hf, eq);</code></td>
<td>Effects: Constructs an empty container with at least <code>n</code> buckets, using <code>hf</code> as the hash function and <code>eq</code> as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>X(n, hf)</code></td>
<td><code>X</code> <code>a(n, hf);</code></td>
<td>Expects: <code>key_equal</code> meets the <code>Cpp17DefaultConstructible</code> requirements. Effects: Constructs an empty container with at least <code>n</code> buckets, using <code>hf</code> as the hash function and <code>key_equal()</code> as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>X(n)</code></td>
<td><code>X</code> <code>a(n);</code></td>
<td>Expects: <code>hasher</code> and <code>key_equal</code> meet the <code>Cpp17DefaultConstructible</code> requirements. Effects: Constructs an empty container with at least <code>n</code> buckets, using <code>hasher()</code> as the hash function and <code>key_equal()</code> as the key equality predicate.</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><code>X()</code></td>
<td><code>X</code> <code>a;</code></td>
<td>Expects: <code>hasher</code> and <code>key_equal</code> meet the <code>Cpp17DefaultConstructible</code> requirements. Effects: Constructs an empty container with an unspecified number of buckets, using <code>hasher()</code> as the hash function and <code>key_equal()</code> as the key equality predicate.</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(i, j, n, hf, eq)</td>
<td>X</td>
<td>Expects: 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 a(i, j, n, hf, eq);</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n, hf)</td>
<td>X</td>
<td>Expects: key_equal meets the Cpp17DefaultConstructible requirements. 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 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 a(i, j, n, hf);</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n)</td>
<td>X</td>
<td>Expects: hasher and key_equal meet the Cpp17DefaultConstructible requirements. value_type is Cpp17EmplaceConstructible into X from *i. Effects: Constructs an empty container with at least n buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j] into it.</td>
<td>Average case $\mathcal{O}(N)$ ($N$ is distance(i, j)), worst case $\mathcal{O}(N^2)$</td>
</tr>
<tr>
<td>X a(i, j, n);</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j)</td>
<td>X</td>
<td>Expects: hasher and key_equal meet the Cpp17DefaultConstructible requirements. value_type is Cpp17EmplaceConstructible into X from *i. Effects: 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 $\mathcal{O}(N)$ ($N$ is distance(i, j)), worst case $\mathcal{O}(N^2)$</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>
</tbody>
</table>
Table 70 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X(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 62, 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 = b</td>
<td>X&amp;</td>
<td>Copy assignment operator. In addition to the requirements of Table 62, 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>Expects: 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>Same as a = X(il).</td>
</tr>
<tr>
<td>b.hash_function()</td>
<td>hasher</td>
<td>Returns: b's hash function.</td>
<td>constant</td>
</tr>
<tr>
<td>b.key_eq()</td>
<td>key_equal</td>
<td>Returns: b's key equality predicate.</td>
<td>constant</td>
</tr>
<tr>
<td>a_uniq.emplace(args)</td>
<td>pair&lt;iterator, bool&gt;</td>
<td>Expects: value_type is Cpp17EmplaceConstructible into X from args. Effects: Inserts a value_type object t constructed with 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.</td>
<td>Average case O(1), worst case O(a_uniq.size()).</td>
</tr>
<tr>
<td>a_eq.emplace(args)</td>
<td>iterator</td>
<td>Expects: value_type is Cpp17EmplaceConstructible into X from args. Effects: Inserts a value_type object t constructed with std::forward&lt;Args&gt;(args)... 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>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.emplace_hint(p, args)</code></td>
<td>iterator</td>
<td><code>Expects: value_type is Cpp17EmplaceConstructible into X from args. Effects: Equivalent to a.emplace(std::forward&lt;Args&gt;(args)...). 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. </code></td>
<td>Average case $O(1)$, worst case $O(a.size())$.</td>
</tr>
<tr>
<td><code>a_uniq.insert(t)</code></td>
<td>pair&lt;iterator, bool&gt;</td>
<td><code>Expects: If t is a non-const rvalue, value_type is Cpp17MoveInsertable into X; otherwise, value_type is Cpp17CopyInsertable into X. Effects: Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of t.</code></td>
<td>Average case $O(1)$, worst case $O(a_uniq.size())$.</td>
</tr>
<tr>
<td><code>a_eq.insert(t)</code></td>
<td>iterator</td>
<td><code>Expects: If t is a non-const rvalue, value_type is Cpp17MoveInsertable into X; otherwise, value_type is Cpp17CopyInsertable into X. Effects: Inserts t, and returns an iterator pointing to the newly inserted element. </code></td>
<td>Average case $O(1)$, worst case $O(a_eq.size())$.</td>
</tr>
<tr>
<td><code>a.insert(p, t)</code></td>
<td>iterator</td>
<td><code>Expects: If t is a non-const rvalue, value_type is Cpp17MoveInsertable into X; otherwise, value_type is Cpp17CopyInsertable into X. Effects: Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of t. The iterator p is a hint pointing to where the search should start. Implementations are permitted to ignore the hint. </code></td>
<td>Average case $O(1)$, worst case $O(a.size())$.</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>a.insert(i, j)</td>
<td>void</td>
<td>Expects: value_type is</td>
<td>Average case (O(N)), where (N) is distance(i, j). Worst case (O(N(a.size() + 1))).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cpp17EmplaceConstructible into (X) from (*i). Neither (i) nor (j) are iterators into (a). Effects: Equivalent to a.insert(t) for each element in ([i,j)).</td>
<td></td>
</tr>
<tr>
<td>a.insert(il)</td>
<td>void</td>
<td>Same as a.insert(il.begin(), il.end()).</td>
<td>Same as a.insert(il.begin(), il.end()).</td>
</tr>
<tr>
<td>a_uniq.insert(nh)</td>
<td>insert_return_type</td>
<td>Expects: (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.\text{key}()). Ensures: If (nh) is empty, (\text{inserted}) is false, (\text{position}) is end(), and (\text{node}) is empty. Otherwise if the insertion took place, (\text{inserted}) is true, (\text{position}) points to the inserted element, and (\text{node}) is empty; if the insertion failed, (\text{inserted}) is false, (\text{node}) has the previous value of (nh), and (\text{position}) points to an element with a key equivalent to (nh.\text{key}()).</td>
<td>Average case (O(1)), worst case (O(a_uniq.size())).</td>
</tr>
<tr>
<td>a_eq.insert(nh)</td>
<td>iterator</td>
<td>Expects: (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. Ensures: (nh) is empty.</td>
<td>Average case (O(1)), worst case (O(a_eq.size())).</td>
</tr>
</tbody>
</table>
Table 70 — Unordered associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.insert(q, nh)</td>
<td>iterator</td>
<td>Expects: nh is empty or a.get_allocator() == nh.get_allocator(). Effects: 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. Ensures: 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>a.extract(k)</td>
<td>node_type</td>
<td>Effects: Removes an element in the container with key equivalent to k. Returns: 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>a.extract(q)</td>
<td>node_type</td>
<td>Effects: Removes the element pointed to by q. Returns: A node_type owning that element.</td>
<td>Average case O(1), worst case O(a.size()).</td>
</tr>
</tbody>
</table>
Table 70 — 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><strong>Expects:</strong> a.get_allocator() == a2.get_allocator(). Attempts to extract each</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>element in a2 and insert it into a using the hash function and key equality</td>
<td>Average case $\mathcal{O}(N)$, where $N$ is a2.size().</td>
</tr>
<tr>
<td></td>
<td></td>
<td>predicate of a. In containers with unique keys, if there is an element in</td>
<td>Worst case $\mathcal{O}(N \times a.size() + N)$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a with key equivalent to the key of an element from a2, then that element is</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>not extracted from a2. <strong>Ensures:</strong> Pointers and references to the transferred</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>elements of a2 refer to those same elements but as members of a. Iterators</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>referring to the transferred elements and all iterators referring to a will</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>be invalidated, but iterators to elements remaining in a2 will remain valid.</td>
<td></td>
</tr>
<tr>
<td>a.erase(k)</td>
<td>size_type</td>
<td><strong>Effects:</strong> Erases all elements with key equivalent to k. <strong>Returns:</strong> The</td>
<td>Average case $\mathcal{O}(a.count(k))$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>number of elements erased.</td>
<td>Worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td><strong>Effects:</strong> Erases the element pointed to by q. <strong>Returns:</strong> The iterator</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>immediately following q prior to the erasure.</td>
<td></td>
</tr>
<tr>
<td>a.erase(r)</td>
<td>iterator</td>
<td><strong>Effects:</strong> Erases the element pointed to by r. <strong>Returns:</strong> The iterator</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>immediately following r prior to the erasure.</td>
<td></td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td><strong>Effects:</strong> Erases all elements in the range $[q1, q2)$. <strong>Returns:</strong> The</td>
<td>Average case linear in $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iterator immediately following the erased elements prior to the erasure.</td>
<td></td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td><strong>Effects:</strong> Erases all elements in the container. <strong>Ensures:</strong> a.empty() is</td>
<td>Linear in $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator:</td>
<td><strong>Returns:</strong> An iterator pointing to an element with key equivalent to k, or</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td></td>
<td>const_iterator for</td>
<td>if no such element exists.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const b.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

§ 22.2.7
Table 70 — 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.find(k, hk)</td>
<td>iterator; const_iterator for const b.</td>
<td><em>Expects:</em> b.hash_function()(k) equals hk. <em>Returns:</em> An iterator pointing to an element with key equivalent to k, or b.end() if no such element exists.</td>
<td>Average case $O(1)$, worst case $O(b.size())$.</td>
</tr>
<tr>
<td>a_tran.find(ke)</td>
<td>iterator; const_iterator for const a_tran.</td>
<td><em>Returns:</em> An iterator pointing to an element with key equivalent to ke, or a_tran.end() if no such element exists.</td>
<td>Average case $O(1)$, worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td>a_tran.find(ke, hke)</td>
<td>iterator; const_iterator for const a_tran.</td>
<td><em>Expects:</em> a_tran.hash_function()(ke) equals hke. <em>Returns:</em> An iterator pointing to an element with key equivalent to ke, or a_tran.end() if no such element exists.</td>
<td>Average case $O(1)$, worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td>b.count(k)</td>
<td>size_type</td>
<td><em>Returns:</em> The number of elements with key equivalent to k.</td>
<td>Average case $O(b.count(k))$, worst case $O(b.size())$.</td>
</tr>
<tr>
<td>b.count(k, hk)</td>
<td>size_type</td>
<td><em>Expects:</em> b.hash_function()(k) equals hk. <em>Returns:</em> The number of elements with key equivalent to k.</td>
<td>Average case $O(b.count(k))$, worst case $O(b.size())$.</td>
</tr>
<tr>
<td>a_tran.count(ke)</td>
<td>size_type</td>
<td><em>Returns:</em> The number of elements with key equivalent to ke.</td>
<td>Average case $O(a_tran.count(ke))$, worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td>a_tran.count(ke, hke)</td>
<td>size_type</td>
<td><em>Expects:</em> a_tran.hash_function()(ke) equals hke. <em>Returns:</em> The number of elements with key equivalent to ke.</td>
<td>Average case $O(a_tran.count(ke))$, worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td>b.contains(k)</td>
<td>bool</td>
<td><em>Effects:</em> Equivalent to b.find(k) != b.end()</td>
<td>Average case $O(1)$, worst case $O(b.size())$.</td>
</tr>
<tr>
<td>b.contains(k, hk)</td>
<td>bool</td>
<td><em>Expects:</em> b.hash_function()(k) equals hk. <em>Effects:</em> Equivalent to b.find(k) != b.end()</td>
<td>Average case $O(1)$, worst case $O(b.size())$.</td>
</tr>
<tr>
<td>a_tran.contains(ke)</td>
<td>bool</td>
<td><em>Effects:</em> Equivalent to a_tran.find(ke) != a_tran.end()</td>
<td>Average case $O(1)$, worst case $O(a_tran.size())$.</td>
</tr>
</tbody>
</table>
Table 70 — 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><code>a_tran.contains(ke, hke)</code></td>
<td>bool</td>
<td>Expects: <code>a_tran.hash_function()(ke)</code> equals <code>hke</code>. Effects: Equivalent to <code>a_tran.find(ke, hke) != a_tran.end()</code></td>
<td>Average case $O(1)$, worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td><code>b.equal_range(k)</code></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 <code>k</code>. Returns <code>make_pair(b.end(), b.end())</code> if no such elements exist.</td>
<td>Average case $O(b.count(k))$. Worst case $O(b.size())$.</td>
</tr>
<tr>
<td><code>b.equal_range(k, hk)</code></td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for const b.</td>
<td>Expects: <code>b.hash_function()(k)</code> equals <code>hk</code>. Returns: A range containing all elements with keys equivalent to <code>k</code>. Returns <code>make_pair(b.end(), b.end())</code> if no such elements exist.</td>
<td>Average case $O(b.count(k))$. Worst case $O(b.size())$.</td>
</tr>
<tr>
<td><code>a_tran.equal_range(ke)</code></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 <code>ke</code>. Returns <code>make_pair(a_tran.end(), a_tran.end())</code> if no such elements exist.</td>
<td>Average case $O(a_tran.count(ke))$. Worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td><code>a_tran.equal_range(ke, hke)</code></td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for const a_tran.</td>
<td>Expects: <code>a_tran.hash_function()(ke)</code> equals <code>hke</code>. Returns: A range containing all elements with keys equivalent to <code>ke</code>. Returns <code>make_pair(a_tran.end(), a_tran.end())</code> if no such elements exist.</td>
<td>Average case $O(a_tran.count(ke))$. Worst case $O(a_tran.size())$.</td>
</tr>
<tr>
<td><code>b.bucket_count()</code></td>
<td>size_type</td>
<td>Returns: The number of buckets that <code>b</code> contains.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.max_bucket_count()</code></td>
<td>size_type</td>
<td>Returns: An upper bound on the number of buckets that <code>b</code> might ever contain.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.bucket(k)</code></td>
<td>size_type</td>
<td>Expects: <code>b.bucket_count() &gt; 0</code>. Returns: The index of the bucket in which elements with keys equivalent to <code>k</code> would be found, if any such element existed. Ensures: The return value shall be in the range $[0, b.bucket_count())$.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.bucket_size(n)</code></td>
<td>size_type</td>
<td>Expects: <code>n</code> shall be in the range $[0, b.bucket_count())$. Returns: The number of elements in the $n^{th}$ bucket.</td>
<td>$O(b.bucket_size(n))$.</td>
</tr>
</tbody>
</table>
### 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><code>b.begin(n)</code></td>
<td><code>local_iterator; const_local_iterator</code> for <code>const b</code></td>
<td><code>Expects: n is in the range [0, b.bucket_count())</code>. Returns: An iterator referring to the first element in the bucket. If the bucket is empty, then <code>b.begin(n) == b.end(n)</code>.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.end(n)</code></td>
<td><code>local_iterator; const_local_iterator</code> for <code>const b</code></td>
<td><code>Expects: n is in the range [0, b.bucket_count())</code>. Returns: An iterator which is the past-the-end value for the bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.cbegin(n)</code></td>
<td><code>const_local_iterator</code></td>
<td><code>Expects: n shall be in the range [0, b.bucket_count())</code>. Returns: An iterator referring to the first element in the bucket. If the bucket is empty, then <code>b.cbegin(n) == b.cend(n)</code>.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.cend(n)</code></td>
<td><code>const_local_iterator</code></td>
<td><code>Expects: n is in the range [0, b.bucket_count())</code>. Returns: An iterator which is the past-the-end value for the bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.load_factor()</code></td>
<td><code>float</code></td>
<td><code>Returns: The average number of elements per bucket.</code></td>
<td>Constant</td>
</tr>
<tr>
<td><code>b.max_load_factor()</code></td>
<td><code>float</code></td>
<td><code>Returns: A positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.</code></td>
<td>Constant</td>
</tr>
<tr>
<td><code>a.max_load_factor(z)</code></td>
<td><code>void</code></td>
<td><code>Expects: z is positive. May change the container's maximum load factor, using z as a hint.</code></td>
<td>Constant</td>
</tr>
<tr>
<td><code>a.reserve(n)</code></td>
<td><code>void</code></td>
<td><code>Same as a.rehash(ceil(n / a.max_load_factor())))</code>.</td>
<td>Average case linear in <code>a.size()</code>, worst case quadratic.</td>
</tr>
</tbody>
</table>

---

12 Two unordered containers `a` and `b` compare equal if `a.size() == b.size()` and, for every equivalent-key group `[Ea1, Ea2)` obtained from `a.equal_range(Ea1)`, there exists an equivalent-key group `[Eb1, Eb2)` obtained from `b.equal_range(Ea1)`, such that `is_permutation(Ea1, Ea2, Eb1, Eb2)` returns true. For `unordered_set` and `unordered_map`, the complexity of `operator==` (i.e., the number of calls to the `==` operator of the `value_type`, to the predicate returned by `key_eq()`, and to the hasher returned by `hash_function()`) is proportional to \( N \) in the average case and to \( N^2 \) in the worst case, where \( N \) is `a.size()`. For `unordered_multiset` and `unordered_multimap`, the complexity of `operator==` is proportional to \( \sum E_i^2 \) in the average case and to \( N^2 \) in the worst case, where \( N \) is `a.size()`, and \( E_i \) is the size of the \( i \)th equivalent-key.
group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups `Ea_i` and `Eb_i` 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 the `Pred` function object has the same behavior for both containers and the equality comparison function for `Key` is a refinement\(^{228}\) of the partition into equivalent-key groups produced by `Pred`.

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.

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.

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.

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.

If the `qualified-id` `Hash::transparent_key_equal` is valid and denotes a type (13.9.2), then the program is ill-formed if either:

(17.1) — `qualified-id` `Hash::transparent_key_equal::is_transparent` is not valid or does not denote a type, or

(17.2) — `Pred` is a different type than `equal_to<Key>` or `Hash::transparent_key_equal`.

The member function templates `find`, `count`, `equal_range`, and `contains` shall not participate in overload resolution unless the `qualified-id` `Hash::transparent_key_equal` is valid and denotes a type (13.9.2).

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

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).

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.

For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the swap of the container's `Hash` or `Pred` object (if any).

For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container's hash function or comparison function, the `rehash()` function has no effect.

---

\(^{228}\) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.


22.3 Sequence containers

22.3.1 In general

The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define class templates that meet the requirements for sequence containers.

The following exposition-only alias template may appear in deduction guides for sequence containers:

```cpp
template<class InputIterator>
using iter_value_type = typename iterator_traits<InputIterator>::value_type;  // exposition only
```

22.3.2 Header `<array>` synopsis

```cpp
#include <initializer_list>
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 bool operator!=(const array<T, N>& x, const array<T, N>& y);
    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 bool operator> (const array<T, N>& x, const array<T, N>& y);
    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 bool operator>=(const array<T, N>& x, const array<T, N>& y);
    template<class T, size_t N> constexpr void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));
    // 22.3.7.6, tuple interface to class template array
    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

```cpp
#include <initializer_list>
namespace std {
    // 22.3.8, class template deque
    template<class T, class Allocator = allocator<T>> class deque;
    template<class T, class Allocator> bool operator==(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template<class T, class Allocator> bool operator!=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template<class T, class Allocator> bool operator< (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template<class T, class Allocator> bool operator> (const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template<class T, class Allocator> bool operator<=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
    template<class T, class Allocator> bool operator>=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);
}
```
template<class T, class Allocator>
bool operator>=(const deque<T, Allocator>& x, const deque<T, Allocator>& y);

template<class T, class Allocator>
bool operator>(const deque<T, Allocator>& x, const deque<T, Allocator>& y);

template<class T, class Allocator>
bool operator<(const deque<T, Allocator>& x, const deque<T, Allocator>& y);

template<class T, class Allocator>
void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
  noexcept(noexcept(x.swap(y)));

template<class T, class Allocator, class U>
void erase(deque<T, Allocator>& c, const U& value);

template<class T, class Allocator, class Predicate>
void erase_if(deque<T, Allocator>& c, Predicate pred);

namespace pmr {
  template<class T>
  using deque = std::deque<T, polymorphic_allocator<T>>;
}

22.3.4 Header <forward_list> synopsis

#include <initializer_list>

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>
  bool operator!=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);

  template<class T, class Allocator>
  bool operator< (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);

  template<class T, class Allocator>
  bool operator> (const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);

  template<class T, class Allocator>
  bool operator>=(const forward_list<T, Allocator>& x, const forward_list<T, Allocator>& y);

  template<class T, class Allocator>
  bool 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>
  void erase(forward_list<T, Allocator>& c, const U& value);

  template<class T, class Allocator, class Predicate>
  void 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 <initializer_list>

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>
  bool operator!=(const list<T, Allocator>& x, const list<T, Allocator>& y);

  template<class T, class Allocator>
  bool operator< (const list<T, Allocator>& x, const list<T, Allocator>& y);

  template<class T, class Allocator>
  bool operator> (const list<T, Allocator>& x, const list<T, Allocator>& y);

  template<class T, class Allocator>
  bool operator>=(const list<T, Allocator>& x, const list<T, Allocator>& y);

  template<class T, class Allocator>
  bool 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>
  void erase(list<T, Allocator>& c, const U& value);

  template<class T, class Allocator, class Predicate>
  void erase_if(list<T, Allocator>& c, Predicate pred);

  namespace pmr {
    template<class T>
      using list = std::list<T, polymorphic_allocator<T>>;
  }
}
template<class T, class Allocator>
    bool operator==(const list<T, Allocator>& x, const list<T, Allocator>& y);

template<class T, class Allocator>
    bool operator!=(const list<T, Allocator>& x, const list<T, Allocator>& y);

template<class T, class Allocator>
    bool operator<( const list<T, Allocator>& x, const list<T, Allocator>& y);

template<class T, class Allocator>
    bool operator>( const list<T, Allocator>& x, const list<T, Allocator>& y);

template<class T, class Allocator>
    bool operator<=(const list<T, Allocator>& x, const list<T, Allocator>& y);

template<class T, class Allocator>
    bool 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>
    void erase(list<T, Allocator>& c, const U& value);

template<class T, class Allocator, class Predicate>
    void erase_if(list<T, Allocator>& c, Predicate pred);

namespace pmr {
    template<class T>
        using list = std::list<T, polymorphic_allocator<T>>;
}

22.3.6 Header <vector> synopsis [vector.syn]

#include <initializer_list>

namespace std {
    // 22.3.11, class template vector
    template<class T, class Allocator = allocator<T>> class vector;

    template<class T, class Allocator>
        bool operator==(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template<class T, class Allocator>
        bool operator!=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template<class T, class Allocator>
        bool operator<( const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template<class T, class Allocator>
        bool operator>( const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template<class T, class Allocator>
        bool operator<=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);
    template<class T, class Allocator>
        bool operator>=(const vector<T, Allocator>& x, const vector<T, Allocator>& y);

    template<class T, class Allocator>
        void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
        noexcept(noexcept(x.swap(y)));

    template<class T, class Allocator, class U>
        void erase(vector<T, Allocator>& c, const U& value);
    template<class T, class Allocator, class Predicate>
        void erase_if(vector<T, Allocator>& c, Predicate pred);

    // 22.3.12, class vector<bool>
    template<class Allocator> class vector<bool, Allocator>;

    // hash support
    template<class T> struct hash;
    template<class Allocator> struct hash<vector<bool, Allocator>>;

§ 22.3.6
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 (9.3.1) that can be list-initialized with up to `N` elements whose types are convertible to `T`.

3. An `array` satisfies 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` satisfies 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. The types `iterator` and `const_iterator` satisfy 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;
    };
}
22.3.7.2 Constructors, copy, and assignment

The conditions for an aggregate (9.3.1) shall be met. Class `array` relies on the implicitly-declared special member functions (11.3.4.1, 11.3.6, and 11.3.4.2) 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.

22.3.7.3 Member functions

22.3.7.4 Specialized algorithms

22.3.7.5 Zero-sized arrays
Member function \texttt{swap}() shall have a non-throwing exception specification.

22.3.7.6 Tuple interface

\begin{verbatim}
template<class T, size_t N>
struct tuple_size<array<T, N>> : integral_constant<size_t, N> { };

tuple_element<I, array<T, N>>::type
\end{verbatim}

\begin{enumerate}
\item \textit{Mandates:} \( I < N \) is true.
\item \textit{Value:} The type \( T \).
\item \textit{Mandates:} \( I < N \) is true.
\end{enumerate}

\textit{Returns:} A reference to the \( I \)th element of \( a \), where indexing is zero-based.

22.3.8 Class template \texttt{deque}

22.3.8.1 Overview

\begin{enumerate}
\item A \texttt{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.
\item A \texttt{deque} satisfies 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 65). Descriptions are provided here only for operations on \texttt{deque} that are not described in one of these tables or for operations where there is additional semantic information.
\end{enumerate}

\begin{verbatim}
namespace std {
  template<class T, class Allocator = allocator<T>>
  class deque {
    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>;

      // 22.3.8.2, 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& x);
      deque(deque&&);
      deque(const deque&, const Allocator&);
      deque(deque&, const Allocator&);

  }; // deque
}
\end{verbatim}
deque(initializer_list<T>, const Allocator& = Allocator());
-deque();
deque& operator=(const deque& x);
deque& operator=(deque&& x)
    noexcept(allocator_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&);
1 Effects: Constructs an empty deque, using the specified allocator.
2 Complexity: Constant.

explicit deque(size_type n, const Allocator& = Allocator());
3 Effects: Constructs a deque with n default-inserted elements using the specified allocator.
4 Expects: T is Cpp17DefaultInsertable into *this.
5 Complexity: Linear in n.

deque(size_type n, const T& value, const Allocator& = Allocator());
6 Effects: Constructs a deque with n copies of value, using the specified allocator.
7 Expects: T is Cpp17CopyInsertable into *this.
8 Complexity: Linear in n.

template<class InputIterator>
deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
9 Effects: Constructs a deque equal to the range [first, last), using the specified allocator.
10 Complexity: Linear in distance(first, last).

22.3.8.3 Capacity [deque.capacity]

void resize(size_type sz);
1 Expects: T is Cpp17MoveInsertable and Cpp17DefaultInsertable into *this.
2 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);
3 Expects: T is Cpp17CopyInsertable into *this.
4 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();
5 Expects: T is Cpp17MoveInsertable into *this.
6 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
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-\texttt{Cpp17CopyInsertable} \(T\), then there are no effects.

7

\textit{Complexity:} If the size is not equal to the old capacity, linear in the size of the sequence; otherwise constant.

8

\textit{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.

\section*{22.3.8.4 Modifiers [deque.modifiers]}

\begin{verbatim}
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);
\end{verbatim}

\subsection*{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.

\subsection*{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-\texttt{Cpp17CopyInsertable} \(T\), the effects are unspecified.

\subsection*{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 \(T\).

\begin{verbatim}
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();
\end{verbatim}

\subsection*{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.

\textit{Note:} \texttt{pop_front} and \texttt{pop_back} are erase operations. —end note

\subsection*{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.

\subsection*{Throws:} Nothing unless an exception is thrown by the assignment operator of \(T\).

\section*{22.3.8.5 Erasure [deque.erasure]}

\begin{verbatim}
template<class T, class Allocator, class U>
    void erase(deque<T, Allocator>& c, const U& value);
template<class T, class Allocator, class Predicate>
    void erase_if(deque<T, Allocator>& c, Predicate pred);
\end{verbatim}

\subsection*{Effects:} Equivalent to: \texttt{c.erase(remove(c.begin(), c.end(), value), c.end());}

\subsection*{Effects:} Equivalent to: \texttt{c.erase(remove_if(c.begin(), c.end(), pred), 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` satisfies all of the requirements of a container (Table 62), except that the `size()` member function is not provided and `operator==` has linear complexity. A `forward_list` also satisfies all of the requirements for an allocator-aware container (Table 65). In addition, a `forward_list` provides the `assign` member functions (Table 66) and several of the optional container requirements (Table 67). 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]

```cpp
namespace std {
    template<class T, class Allocator = allocator<T>>
    class forward_list {
    public:
        // types
        using value_type = T;
        using allocator_type = Allocator;
        using pointer = typename allocator_traits<Allocator>::pointer;
        using const_pointer = typename allocator_traits<Allocator>::const_pointer;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = implementation-defined; // see 22.2
        using difference_type = implementation-defined; // see 22.2
        using iterator = implementation-defined; // see 22.2
        using const_iterator = implementation-defined; // see 22.2

        // 22.3.9.2, construct/copy/destroy
        forward_list() : forward_list(Allocator()) { }
        explicit forward_list(const Allocator&);
        explicit forward_list(size_type n, const Allocator& = Allocator());
        forward_list(size_type n, const T& value, const Allocator& = Allocator());
        template<class InputIterator>
            forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
        forward_list(const forward_list& x);
        forward_list(forward_list&& x);
        forward_list(const forward_list& x, const Allocator&);
        forward_list(forward_list&& x, const Allocator&);
        forward_list(initializer_list<T>, const Allocator& = Allocator());
        ~forward_list();
        forward_list& operator=(const forward_list& x);
        forward_list& operator=(forward_list&& x) noexcept(allocator_traits<Allocator>::is_always_equal::value);
        forward_list& operator=(initializer_list<T>);
        template<class InputIterator>
            void assign(InputIterator first, InputIterator last);
        void assign(size_type n, const T& t);
        void assign(initializer_list<T>);
        allocator_type get_allocator() const noexcept;

        // 22.3.9.3, iterators
        iterator before_begin() noexcept;
        const_iterator before_begin() const noexcept;
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        iterator end() const noexcept;
    }
}
```

§ 22.3.9.1 805
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;

§ 22.3.9.1
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 satisfies 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

#### [forwardlist.cons]

**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. **Expects:** 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. **Expects:** 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>**

9. **forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());**

10. **Effects:** Constructs a forward_list object equal to the range [first, last).
11. **Complexity:** Linear in distance(first, last).

### 22.3.9.3 Iterators

#### [forwardlist.iter]

**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

#### [forwardlist.access]

**reference front();**
**const_reference front() const;**

1. **Returns:** *begin()

### 22.3.9.5 Modifiers

#### [forwardlist.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);

Effects: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
Returns: An iterator pointing to the copy of x.

iterator insert_after(const_iterator position, size_type n, const T& x);

Effects: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
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);

Effects: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
Neither first nor last are iterators in *this.
Returns: 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);

Effects: 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);

Effects: The iterator following position is dereferenceable.
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);

Effects: 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);

   Expects: 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);

   Expects: 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;

   Effects: Erases all elements in the range [begin(), end()).
   Remarks: Does not invalidate past-the-end iterators.

22.3.9.6 Operations [forwardlist.ops]

In this subclause, arguments for a template parameter named Predicate or BinaryPredicate shall satisfy the corresponding requirements in 25.2. For merge and sort, the definitions and requirements in 25.7 apply.

void splice_after(const_iterator position, forward_list& x);
void splice_after(const_iterator position, forward_list&& x);

   Expects: 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: O(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);

   Expects: 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)

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);

   Expects: 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: $O(distance(first, last))$

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 (16.5.5.7).

Complexity: Exactly distance(begin(), end()) applications of the corresponding predicate.

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.

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);

Expects: *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 (16.5.5.7).

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 (16.5.5.7).

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>
void erase(forward_list<T, Allocator>& c, const U& value);

Effects: Equivalent to: erase_if(c, [&](auto& elem) return elem == value; );

template<class T, class Allocator, class Predicate>
void erase_if(forward_list<T, Allocator>& c, Predicate pred);

Effects: Equivalent to: c.remove_if(pred);

22.3.10 Class template `list`

22.3.10.1 Overview

1 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.

2 A `list` satisfies 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 65). The exceptions are the `operator[]` and `at` member functions, which are not provided. 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.

```cpp
namespace std {
    template<class T, class Allocator = allocator<T>>
    class list {
public:
    // types
    using value_type = T;
    using allocator_type = Allocator;
    using pointer = typename allocator_traits<Allocator>::pointer;
    using const_pointer = typename allocator_traits<Allocator>::const_pointer;
    using reference = value_type&;
    using const_reference = const value_type&;
    using size_type = implementation-defined;  // see 22.2
    using difference_type = implementation-defined;  // see 22.2
    using iterator = implementation-defined;  // see 22.2
    using const_iterator = implementation-defined;  // see 22.2
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 22.3.10.2, construct/copy/destroy
    list() : list(Allocator()) { }  
    explicit list(const Allocator&);  
    explicit list(size_type n, const Allocator& = Allocator());  
    list(size_type n, const T& value, const Allocator& = Allocator());  
    template<class InputIterator>
    list(InputIterator first, InputIterator last, const Allocator& = Allocator());  
    template<class InputIterator>
    list(InputIterator first, InputIterator last, const Allocator& = Allocator());  
    list(const list& x);  
    list(const list&, const Allocator&);  
    list(list&& x);  
    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;
};
```

229) These member functions are only provided by containers whose iterators are random access iterators.

§ 22.3.10.1
// 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;
};

template<class InputIterator, class Allocator = allocator<
    iter-value-type<InputIterator>>>>
    list(InputIterator, InputIterator, Allocator = Allocator())
        -> list<
    iter-value-type<InputIterator>, Allocator>;

// swap
    template<class T, class Allocator>
        void swap(list<T, Allocator>& x, list<T, Allocator>& y)
            noexcept(noexcept(x.swap(y)));

3 An incomplete type T may be used when instantiating list if the allocator satisfies 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

**explicit list(const Allocator&);**
1 *Effects:* Constructs an empty list, using the specified allocator.
2 *Complexity:* Constant.

**explicit list(size_type n, const Allocator& = Allocator());**
3 *Expects:* T is Cpp17DefaultInsertable into *this.
4 *Effects:* Constructs a list with n default-inserted elements using the specified allocator.
5 *Complexity:* Linear in n.

**list(size_type n, const T& value, const Allocator& = Allocator());**
6 *Expects:* T is Cpp17CopyInsertable into *this.
7 *Effects:* Constructs a list with n copies of value, using the specified allocator.
8 *Complexity:* Linear in n.

**template<class InputIterator>**
    list(InputIterator first, InputIterator last, const Allocator& = Allocator());
9 *Effects:* Constructs a list equal to the range [first, last).
10 *Complexity:* Linear in distance(first, last).

### 22.3.10.3 Capacity

**void resize(size_type sz);**
1 *Expects:* T is Cpp17DefaultInsertable into *this.
2 *Effects:* If size() < sz, appends sz - size() default-inserted elements to the sequence. If sz <= size(), equivalent to:
    list<T>::iterator it = begin();
    advance(it, sz);
    erase(it, end());
void resize(size_type sz, const T& c);

   Expects: T is Cpp17CopyInsertable into *this.

   Effects: As if by:

   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

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.

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();
void clear() noexcept;

   Effects: Invalidates only the iterators and references to the erased elements.

   Throws: Nothing.

   Complexity: Erasing a single element is a constant time operation with a single call to the destructor
   of T. Erasing a range in a list is linear time in the size of the range and the number of calls to the
   destructor of type T is exactly equal to the size of the range.

22.3.10.5 Operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically
for them.\footnote{As specified in 16.5.3.5, the requirements in this Clause apply only to lists whose allocators compare equal.}
In this subclause, arguments for a template parameter named Predicate or BinaryPredicate
shall satisfy the corresponding requirements in 25.2. For merge and sort, the definitions and requirements in
25.7 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);

Expects: 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);

Expects: 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);

Expects: [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.

Returns: The number of elements erased.

Throws: Nothing unless an exception is thrown by *i == value or pred(*i) != false.

Remarks: Stable (16.5.5.7).

Complexity: Exactly size() applications of the corresponding predicate.

size_type unique();

template<class BinaryPredicate> size_type unique(BinaryPredicate binary_pred);

Effects: Erases all but the first element from every consecutive group of equal elements referred to by the iterator i in the range [first + 1, last) for which *i == *(i-1) (for the version of unique with no arguments) or pred(*i, *(i - 1)) (for the version of unique with a predicate argument) holds. Invalidates only the iterators and references to the erased elements.

Returns: The number of elements erased.

Throws: Nothing unless an exception is thrown by *i == *(i-1) or pred(*i, *(i - 1))

Complexity: If the range [first, last) is not empty, exactly (last - first) - 1 applications of the corresponding predicate, otherwise no applications of the predicate.

void merge(list& x);

§ 22.3.10.5
void merge(list&& x);

template<class Compare>
void merge(list& x, Compare comp);

template<class Compare>
void merge(list&& x, Compare comp);

Expects: Both the list and the argument list shall be sorted with respect to the comparator operator< (for the first two overloads) or comp (for the last two overloads). get_allocator() != x.get_allocator() is true.

Effects: If addressof(x) == this, does nothing; otherwise, merges the two sorted ranges [begin(), end()) and [x.begin(), x.end()). The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by comp; that is, for every iterator i, in the range other than the first, the condition comp(*i, *(i - 1)) will be false. Pointers and references to the moved elements of x now refer to those same elements but as members of *this. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into *this, not into x.

Remarks: Stable (16.5.5.7). If addressof(x) != this, the range [x.begin(), x.end()) is empty after the merge. No elements are copied by this operation.

Complexity: At most size() + x.size() - 1 applications of comp if addressof(x) != this; otherwise, no applications of comp are performed. If an exception is thrown other than by a comparison there are no effects.

void reverse() noexcept;

Effects: Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

Complexity: Linear time.

void sort();

template<class Compare>
void sort(Compare comp);

Effects: Sorts the list according to the operator< or a Compare function object. If an exception is thrown, the order of the elements in *this is unspecified. Does not affect the validity of iterators and references.

Remarks: Stable (16.5.5.7).

Complexity: Approximately $N \log N$ comparisons, where $N = \text{size}()$.

22.3.10.6 Erasure [list.erasure]

template<class T, class Allocator, class U>
void erase(list<T, Allocator>& c, const U& value);

Effects: Equivalent to: erase_if(c, [&](auto& elem) return elem == value; );

template<class T, class Allocator, class Predicate>
void erase_if(list<T, Allocator>& c, Predicate pred);

Effects: Equivalent to: c.remove_if(pred);

22.3.11 Class template vector [vector]

22.3.11.1 Overview [vector.overview]

A vector is a sequence container that supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency.

A vector satisfies 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 65), 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.
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
            vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
            explicit vector(const Allocator&) noexcept;
            explicit vector(size_type n, const Allocator& = Allocator());
            template<class InputIterator>
            vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
            vector(const vector& x);
            vector(vector&&) noexcept;
            vector(const vector&, const Allocator&);
            vector(vector&&, const Allocator&);
            vector(initializer_list<T>, const Allocator& = Allocator());
            ~vector();
            vector& operator=(const vector& x);
            vector& operator=(vector&& x) noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
                allocator_traits<Allocator>::is_always_equal::value);
            vector& operator=(initializer_list<T>);
            template<class InputIterator>
            void assign(InputIterator first, InputIterator last);
            void assign(size_type n, const T& u);
            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;

            // 22.3.11.3, capacity
            [[nodiscard]] bool empty() const noexcept;
            size_type size() const noexcept;
            size_type max_size() const noexcept;
            size_type capacity() const noexcept;
            void resize(size_type sz);
            void resize(size_type sz, const T& c);
    }
void reserve(size_type n);
void shrink_to_fit();

// element access
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// 22.3.11.4, data access
T* data() noexcept;
const T* data() const noexcept;

// 22.3.11.5, modifiers
template<class... Args> reference emplace_back(Args&&... args);
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);

// 22.3.11.1, 22.3.11.3, and 22.3.11.4

3 An incomplete type \( T \) may be used when instantiating \( \text{vector} \) if the allocator satisfies the allocator completeness requirements (16.5.3.5.1). \( T \) shall be complete before any member of the resulting specialization of \( \text{vector} \) is referenced.

22.3.11.2 Constructors, copy, and assignment

explicit vector(const Allocator&) noexcept;

1 Effects: Constructs an empty \( \text{vector} \), using the specified allocator.

2 Complexity: Constant.

explicit vector(size_type n, const Allocator& = Allocator());

3 Effects: \( T \) is \texttt{Cpp17DefaultInsertable} into \texttt{*this}.

4 Complexity: Linear in \( n \).
vector(size_type n, const T& value, 
const Allocator& = Allocator());

Effects: Constructs a vector with n copies of value, using the specified allocator.

Complexity: Linear in n.

```
template<class InputIterator>
vector(InputIterator first, InputIterator last, 
const Allocator& = Allocator());
```

Effects: Constructs a vector equal to the range [first, last), using the specified allocator.

Complexity: Makes only N calls to the copy constructor of T (where N is the distance between first and last) and no reallocations if iterators first and last are of forward, bidirectional, or random access categories. It makes order N calls to the copy constructor of T and order log N reallocations if they are just input iterators.

22.3.11.3 Capacity

```
size_type capacity() const noexcept;
```

Returns: The total number of elements that the vector can hold without requiring reallocation.

Complexity: Constant time.

```
void reserve(size_type n);
```

Effects: T is Cpp17MoveInsertable into *this.

```
Effects: A directive that informs a vector of a planned change in size, so that it can manage the storage allocation 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(). If an exception is thrown other than by the move constructor of a non-Cpp17CopyInsertable type, there are no effects.
```

Complexity: It does not change the size of the sequence and takes at most linear time in the size of the sequence.

Throws: length_error if n > max_size().

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] No reallocation shall take place during insertions that happen after a call to reserve() until an insertion would make the size of the vector greater than the value of capacity().

```
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. 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]

```
void swap(vector& x)

noexcept(algorithm_traits<Allocator>::propagate_on_container_swap::value || 
algorithm_traits<Allocator>::is_always_equal::value);
```

Effects: Exchanges the contents and capacity() of *this with that of x.

Complexity: Constant time.

---

231) reserve() uses Allocator::allocates() which may throw an appropriate exception.
© ISO/IEC

22.3.11.4 Data

T* data() noexcept;
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

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_back(Args&&... args);
template<class... Args> iterator emplace(const_iterator position, Args&&... args);
void push_back(const T& x);
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.

iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_back();

Effects: Invalidates iterators and references at or after the point of the erase.
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.
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>
void erase(vector<T, Allocator>& c, const U& value);
```

**Effects:** Equivalent to: `c.erase(remove(c.begin(), c.end(), value), c.end());`

```
template<class T, class Allocator, class Predicate>
void erase_if(vector<T, Allocator>& c, Predicate pred);
```

**Effects:** Equivalent to: `c.erase(remove_if(c.begin(), c.end(), pred), c.end());`

22.3.12 Class `vector<bool>`

To optimize space allocation, a specialization of `vector` for `bool` elements is provided:

```
namespace std {
    template<class Allocator>
    class vector<bool, Allocator> {
        public:
            // types
            using value_type = bool;
            using allocator_type = Allocator;
            using pointer = implementation-defined;
            using const_pointer = implementation-defined;
            using const_reference = bool;
            using size_type = implementation-defined;
            using difference_type = implementation-defined;
            using iterator = implementation-defined;
            using const_iterator = implementation-defined;
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;

            // bit reference
            class reference {
                friend class vector;
                reference() noexcept;
                public:
                    reference(const reference&) = default;
                    ~reference();
                    operator bool() const noexcept;
                    reference& operator=(const bool x) noexcept;
                    reference& operator=(const reference& x) noexcept;
                    void flip() noexcept; // flips the bit
            };

            // construct/copy/destroy
            vector() : vector(Allocator()) { }
            explicit vector(const Allocator&);
            explicit vector(size_type n, const Allocator& = Allocator());
            vector(size_type n, const bool& value, const Allocator& = Allocator());
            template<class InputIterator>
            vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
            vector(const vector& x);
            vector(vector&& x);
            vector(const vector&, const Allocator&);
            vector(vector&, const Allocator&);
            vector(initializer_list<bool>, const Allocator& = Allocator());
            ~vector();
            vector& operator=(const vector& x);
            vector& operator=(vector&& x);
            vector& operator=(initializer_list<bool>);
            template<class InputIterator>
            void assign(InputIterator first, InputIterator last);
            void assign(size_type n, const bool& t);
            void assign(initializer_list<bool>);
            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;
size_type capacity() const noexcept;
void resize(size_type sz, bool c = false);
void reserve(size_type n);
void shrink_to_fit();

// element access
reference operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference at(size_type n);
reference front();
const_reference front() const;
reference back();
const_reference back() const;

// modifiers
template<class... Args> reference emplace_back(Args&&... args);
void push_back(const bool& x);
void pop_back();
template<class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const bool& x);
iterator insert(const_iterator position, size_type n, const bool& x);
template<class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, initializer_list<bool> il);
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void swap(vector&);
static void swap(reference x, reference y) noexcept;
void flip() noexcept; // flips all bits
void clear() noexcept;

2 Unless described below, all operations have the same requirements and semantics as the primary vector template, except that operations dealing with the bool value type map to bit values in the container storage and allocator_traits::construct (20.10.9.2) 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.
void flip() noexcept;

Effects: Replaces each element in the container with its complement.

static void swap(reference x, reference y) noexcept;

Effects: Exchanges the contents of x and y as if by:

```cpp
bool b = x;
x = y;
y = b;
```

template<class Allocator> struct hash<vector<bool, Allocator>>;

The specialization is enabled (20.14.18).

22.4 Associative containers

22.4.1 In general

The header `<map>` defines the class templates `map` and `multimap`; the header `<set>` defines the class templates `set` and `multiset`.

The following exposition-only alias templates may appear in deduction guides for associative containers:

```cpp
template<class InputIterator>
using iter-value-type =
    typename iterator_traits<InputIterator>::value_type; // exposition only

template<class InputIterator>
using iter-key-type =
    remove_const_t<
        typename iterator_traits<InputIterator>::value_type::first_type>; // exposition only

template<class InputIterator>
using iter-mapped-type =
    typename iterator_traits<InputIterator>::value_type::second_type; // exposition only

template<class InputIterator>
using iter-to-alloc-type =
    pair<
        add_const_t<typename iterator_traits<InputIterator>::value_type::first_type>,
        typename iterator_traits<InputIterator>::value_type::second_type>; // exposition only
```

22.4.2 Header `<map>` synopsis

```cpp
#include <initializer_list>

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>
    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>
    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>
    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>
    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>
    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>
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>
void 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>
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>
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>
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>
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>
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>
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>
void 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>>;}

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>
bool operator!=(const set<Key, Compare, Allocator>& x,
const set<Key, Compare, Allocator>& y);

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>
bool operator> (const set<Key, Compare, Allocator>& x,
const set<Key, Compare, Allocator>& y);

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>
bool 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>
void swap(set<Key, Compare, Allocator>& x,
set<Key, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));

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>>;
}

// 22.4.7, 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>
bool operator!=(const multiset<Key, Compare, Allocator>& x,
const multiset<Key, Compare, Allocator>& y);

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>
bool operator> (const multiset<Key, Compare, Allocator>& x,
const multiset<Key, Compare, Allocator>& y);

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>
bool 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>
void erase_if(multiset<Key, Compare, Allocator>& c, Predicate pred);

amespace 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>>;
}

§ 22.4.3
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 satisfies 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 65). 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.

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());

§ 22.4.4.1
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
pair<iterator, bool> emplace(Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
pair<iterator, bool> insert(P&& x);
pair<iterator, bool> insert(const_iterator position, const value_type& x);
pair<iterator, bool> insert(const_iterator position, value_type&& x);
pair<iterator, bool> insert(const_iterator position, P&& x);
pair<iterator, bool> insert(const_iterator position, const value_type& x);
pair<iterator, bool> insert(const_iterator position, value_type&& x);
pair<iterator, bool> insert(const_iterator hint, node_type&& nh);
pair<iterator, bool> try_emplace(const key_type& k, Args&&... 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);

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);

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) const;
template<class InputIterator, class Compare = less<typename InputIterator::value_type>,
class Allocator = allocator<typename InputIterator::value_type>, Compare = Compare(), Allocator = Allocator()>
map(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
-> map<typename InputIterator::value_type, typename InputIterator::value_type, Compare, Allocator>;

template<class Key, class T, class Compare = less<Key>,
class Allocator = allocator<std::pair<const Key, T>>, Compare = Compare(), Allocator = Allocator()>
map(initializer_list<std::pair<Key, T>>, Compare = Compare(), Allocator = Allocator())
-> map<Key, T, Compare, Allocator>;

template<class InputIterator, class Allocator>
map(InputIterator, InputIterator, Allocator) 
-> map<typename InputIterator::value_type, typename InputIterator::value_type, less<typename InputIterator::value_type>, Allocator>;

template<class Key, class T, class Allocator>
map(initializer_list<std::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);
template<class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
```

Expects: value_type is Cpp17EmplaceConstructible into map from piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...).

Effects: If the map already contains an element whose key is equivalent to k, there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...). Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

```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);
```

Expects: value_type is Cpp17EmplaceConstructible into map from piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...).

Effects: If the map already contains an element whose key is equivalent to k, there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...). Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

```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.

Expects: value_type is Cpp17EmplaceConstructible into map from k, forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with k, std::forward<M>(obj). Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

```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.

Expects: value_type is Cpp17EmplaceConstructible into map from move(k), forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with std::move(k), std::forward<M>(obj). Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.
22.4.4.5 Erasure

```
template<class Key, class T, class Compare, class Allocator, class Predicate>
void erase_if(map<Key, T, Compare, Allocator>& c, Predicate pred);
```

1. Effects: Equivalent to:
   ```
   for (auto i = c.begin(), last = c.end(); i != last; ) {
     if (pred(*i)) {
       i = c.erase(i);
     } else {
       ++i;
     }
   }
   ```

22.4.5 Class template multimap

22.4.5.1 Overview

A `multimap` is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys. The `multimap` class supports bidirectional iterators.

A `multimap` satisfies 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 65). A `multimap` also provides most operations described in 22.2.6 for equal keys. This means that a `multimap` supports the `a_eq` operations in 22.2.6 but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`.

Descriptions are provided here only for operations on `multimap` that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template<class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class multimap {
    public:
      // types
      using key_type = Key;
      using mapped_type = T;
      using value_type = pair<const Key, T>;
      using key_compare = Compare;
      using allocator_type = Allocator;
      using pointer = typename allocator_traits<Allocator>::pointer;
      using const_pointer = typename allocator_traits<Allocator>::const_pointer;
      using reference = value_type&;
      using const_reference = const value_type&;
      using size_type = implementation-defined; // see 22.2
      using difference_type = implementation-defined; // see 22.2
      using iterator = implementation-defined; // see 22.2
      using const_iterator = implementation-defined; // see 22.2
      using reverse_iterator = std::reverse_iterator<iterator>;
      using const_reverse_iterator = std::reverse_iterator<const_iterator>;
      using node_type = unspecified;
      class value_compare {
        friend class multimap;
        protected:
          Compare comp;
        value_compare(Compare c) : comp(c) { }
        public:
          bool operator()(const value_type& x, const value_type& y) const {
            return comp(x.first, y.first);
          }
      };

    // 22.4.5.2, construct/copy/destroy
    multimap() : multimap(Compare()) { }
    explicit multimap(const Compare& comp, const Allocator& = Allocator());
  };
```
template<class InputIterator>
    multimap(InputIterator first, InputIterator last,
             const Compare& comp = Compare(),
             const Allocator& = Allocator());
multimap(const multimap& x);
multimap(multimap&& x);
explicit multimap(const Allocator&);
multimap(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(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_move_assignable_v<Compare>);
multimap& operator=(initializer_list<value_type> il, const Allocator& a);
    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&);

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 InputIterator, class Allocator>
multimap(InputIterator, InputIterator, Allocator)
  -> multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
              less<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
multimap(initializer_list<pair<Key, T>>, Allocator)
  -> multimap<Key, T, less<Key>, Allocator>;

// swap
template<class Key, class T, class Compare, class Allocator>
void swap(multimap<Key, T, Compare, Allocator>& x,
          multimap<Key, T, Compare, Allocator>& y)
  noexcept(noexcept(x.swap(y)));

22.4.5.2 Constructors

explicit multimap(const Compare& comp, const Allocator& = Allocator());

1 Effects: Constructs an empty multimap using the specified comparison object and allocator.
2 Complexity: Constant.

template<class InputIterator>
multimap(InputIterator first, InputIterator last,
         const Compare& comp = Compare(),
         const Allocator& = Allocator());
3 Effects: Constructs an empty multimap 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 \( \text{comp} \) and otherwise
  \( N \log N \), where \( N \) is \( \text{last}-\text{first} \).

22.4.5.3 Modifiers

template<class P> iterator insert(P&& x);

1 Mandates: \text{is_constructible}_{V<\text{value_type}, P&&>} is true.
2 Effects: The first form is equivalent to return \( \text{emplace} \)(std::forward<P>(x)). The second form is
  equivalent to return \( \text{emplace}_\text{hint} \)(position, std::forward<P>(x)).

22.4.5.4 Erasure

template<class Key, class T, class Compare, class Allocator, class Predicate>
void erase_if(multimap<Key, T, Compare, Allocator>& c, Predicate pred);
1 Effects: Equivalent to:
  for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
      i = c.erase(i);
    } else {
      ++i;
    }
  }

22.4.6 Class template set

22.4.6.1 Overview

1 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.
2 A set satisfies 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 65). A set also provides most operations
  described in 22.2.6 for unique keys. This means that a set supports the \text{a}_\text{uniq} operations in 22.2.6 but
  not the \text{a}_\text{eq} operations. For a set<Key> both the \text{key_type} and \text{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.

```cpp
namespace std {
    template<class Key, class Compare = less<Key>,
        class Allocator = allocator<Key>>
    class set {
        public:
            // types
            using key_type = Key;
            using key_compare = Compare;
            using value_type = Key;
            using value_compare = Compare;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 22.2
            using difference_type = implementation-defined; // see 22.2
            using iterator = implementation-defined; // see 22.2
            using const_iterator = implementation-defined; // see 22.2
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;
            using node_type = unspecified;
            using insert_return_type = insert-return-type<iterator, node_type>;

            // 22.4.6.2, construct/copy/destroy
            set() : set(Compare()) { }
            explicit set(const Compare& comp, const Allocator& = Allocator());
            template<class InputIterator>
            set(InputIterator first, InputIterator last,
                const Compare& comp = Compare(), const Allocator& = Allocator());
            set(const set& x);
            set(set&& x);
            explicit set(const Allocator&);
            set(const set&, const Allocator&);
            set(const Allocator&, const 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_move_assignable_v<Compare>);
            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;
    }
}
```

§ 22.4.6.1 835
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(set&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_swappable_v<Compare>);
void clear() noexcept;

template<class C2>
    void merge(set<Key, C2, Allocator>& source);
template<class C2>
    void merge(set<Key, C2, Allocator>&& source);
template<class C2>
    void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
    void merge(multiset<Key, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// set operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;

size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

bool contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

§ 22.4.6.1
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;

pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K> pair<iterator, iterator> equal_range(const K& x);
template<class K> pair<const_iterator, const_iterator> equal_range(const K& x) const;

};

template<class InputIterator,
class Compare = less<
iter_value_type
<InputIterator>>,
class Allocator = allocator<
iter_value_type
<InputIterator>>>
set(InputIterator, InputIterator,
Compare = Compare(), Allocator = Allocator())
-> set<
iter_value_type
<InputIterator>, Compare, Allocator>;

template<class Key, class Compare = less<Key>,
class Allocator = allocator<Key>>
set(initializer_list<Key>, Compare = Compare(), Allocator = Allocator())
-> set<Key, Compare, Allocator>;

template<class InputIterator, class Allocator>
set(InputIterator, InputIterator, Allocator)
-> set<
iter_value_type
<InputIterator>,
less<
iter_value_type
<InputIterator>>, Allocator>;

template<class Key, class Allocator>
set(initializer_list<Key>, Allocator) -> set<Key, less<Key>, Allocator>;

// swap
template<class Key, class Compare, class Allocator>
void swap(set<Key, Compare, Allocator>& x,
set<Key, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));

22.4.6.2 Constructors, copy, and assignment

explicit set(const Compare& comp, const Allocator& = Allocator());

Effects: Constructs an empty set using the specified comparison objects and allocator.

Complexity: Constant.

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).

Complexity: Linear in N if the range [first, last) is already sorted using comp and otherwise
N log N, where N is last - first.

22.4.6.3 Erasure

template<class Key, class Compare, class Allocator, class Predicate>
void erase_if(set<Key, Compare, Allocator>& c, Predicate pred);

Effects: Equivalent to:

for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
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 satisfies 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 65). multiset also provides most operations described in 22.2.6 for duplicate keys. This means that a multiset supports the a_eq operations in 22.2.6 but not the a_uniq operations. For a multiset<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on multiset that are not described in one of these tables and for operations where there is additional semantic information.

namespace std {
    template<class Key, class Compare = less<Key>,
             class Allocator = allocator<Key>>
    class multiset {
        public:
            // types
            using key_type = Key;
            using key_compare = Compare;
            using value_type = Key;
            using value_compare = Compare;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 22.2
            using difference_type = implementation-defined; // see 22.2
            using iterator = implementation-defined; // see 22.2
            using const_iterator = implementation-defined; // see 22.2
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;
            using node_type = unspecified;

            // 22.4.7.2, construct/copy/destroy
            multiset() : multiset(Compare()) { }
            explicit multiset(const Compare& comp, const Allocator& = Allocator());
            template<class InputIterator>
            multiset(InputIterator first, InputIterator last,
                      const Compare& comp = Compare(), const Allocator& = Allocator());
            multiset(const multiset& x);
            multiset(multiset&& x);
            explicit multiset(const Allocator&);
            multiset(const multiset&, const Allocator&);
            multiset(multiset&, const Allocator&);
            multiset(initializer_list<value_type>, const Compare& = Compare(),
                     const Allocator& = Allocator());
            template<class InputIterator>
            multiset(InputIterator first, InputIterator last, const Allocator& a)
            : multiset(first, last, Compare(), a) { }
            multiset(initializer_list<value_type> il, const Allocator& a)
            : multiset(il, Compare(), a) { }
            ~multiset();
            multiset& operator=(const multiset& x);
            multiset& operator=(multiset&& x)
            noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                      is_nothrow_move_assignable_v<Compare>);
    }

§ 22.4.7.1 838
multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;

reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;

const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);

iterator insert(const value_type& x);
iterator insert(value_type&& x);

iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);

template<class InputIterator>
void insert(InputIterator first, InputIterator last);

node_type extract(const_iterator position);
node_type extract(const key_type& x);

iterator insert(node_type&& nh);

iterator erase(iterator position);
iterator erase(const_iterator position);

size_type erase(const key_type& x);

iterator erase(const_iterator first, const_iterator last);

void swap(multiset&)
noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Compare>);

void clear() noexcept;

template<class C2>
void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
void merge(multiset<Key, C2, Allocator>&& source);
template<class C2>
void merge(set<Key, C2, Allocator>& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// set operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;

size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

bool contains(const key_type& x) const;
template<class K> bool contains(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;

pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K> pair<iterator, iterator> equal_range(const K& x);
template<class K> pair<const_iterator, const_iterator> equal_range(const K& x) const;

};

template<class InputIterator, 
class Compare = less<
iter_value_type<InputIterator>>, 
class Allocator = allocator<
iter_value_type<InputIterator>>> 
multiset(InputIterator, InputIterator, 
Compare = Compare(), Allocator = Allocator()) 
-> multiset<
iter_value_type<InputIterator>, Compare, Allocator>;

template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>> 
multiset(initializer_list<Key>, Compare = Compare(), Allocator = Allocator()) 
-> multiset<Key, Compare, Allocator>;

template<class InputIterator, class Allocator> 
multiset(InputIterator, InputIterator, Allocator) 
-> multiset<
iter_value_type<InputIterator>, 
less<
iter_value_type<InputIterator>>, Allocator>;

template<class Key, class Allocator> 
multiset(initializer_list<Key>, Allocator) 
-> multiset<Key, less<Key>, Allocator>;

// swap 
void swap(multiset<Key, Compare, Allocator>& x, 
multiset<Key, Compare, Allocator>& y) 
noexcept(noexcept(x.swap(y)));

22.4.7.2 Constructors

explicit multiset(const Compare& comp, const Allocator& = Allocator());

Effects: Constructs an empty multiset using the specified comparison object and allocator.

Complexity: Constant.
const Compare& comp = Compare(), const Allocator& = Allocator());

Effects: Constructs an empty multiset 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.

22.4.7.3 Erasure

template<class Key, class Compare, class Allocator, class Predicate>
void erase_if(multiset<Key, Compare, Allocator>& c, Predicate pred);

Effects: Equivalent to:

```cpp
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}
```

22.5 Unordered associative containers

22.5.1 In general

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.

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 <initializer_list>

namespace std {
    // 22.5.4, class template unordered_map
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>,
             class Alloc = allocator<pair<const Key, T>>>
    class unordered_map;

    // 22.5.5, class template unordered_multimap
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>,
             class Alloc = allocator<pair<const Key, T>>>
    class unordered_multimap;

    template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator==(const unordered_map<Key, T, Hash, Pred, Alloc>& a,
                    const unordered_map<Key, T, Hash, Pred, Alloc>& b);

    template<class Key, class T, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_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>
    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 Key, class T, class H, class P, class A, class Predicate>
void erase_if(unordered_map<Key, T, H, P, A>& c, Predicate pred);

template<class Key, class T, class H, class P, class A, class Predicate>
void erase_if(unordered_multimap<Key, T, H, P, A>& c, Predicate pred);

namespace pmr {
    template<class Key, class T, class Hash = hash<Key>, class Pred = equal_to<Key>>
    using unordered_map
        = std::unordered_map<Key, T, Hash, Pred,
                            polymorphic_allocator<pair<const Key, T>>;}

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_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);

§ 22.5.3
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 satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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>.

This subclause only describes operations on unordered_map that are not described in one of the requirement tables, or for which there is additional semantic information.
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&);
exPLICIT 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 hasher& hf, const allocator_type& a)
  : unordered_map(n, hf, key_equal(), a) { }

template<class InputIterator>
unordered_map(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
  : unordered_map(f, l, n, hasher(), key_equal(), a) { }

unordered_map(initializer_list<value_type> il, size_type n, const allocator_type& a)
  : unordered_map(il, n, hasher(), key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const hasher& hf,
    const allocator_type& a)
  : unordered_map(il, n, hf, key_equal(), a) { }
~unordered_map();
unordered_map& operator=(const unordered_map&);
unordered_map& operator=(unordered_map&&)
    noexcept(allocator_traits<Allocator>:::is_always_equal::value &&k
    is_nothrow_move_assignable_v<Hash> &&k
    is_nothrow_move_assignable_v<Pred>);
unordered_map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 22.5.4.4, modifiers
template<
class... Args>
pair<
iterator, bool>
emplace(Args&&... args);

template<
class... Args>
iterator
emplace_hint(const_iterator position, Args&&... args);
pair<
iterator, bool>
insert(const value_type& obj);
pair<
iterator, bool>
insert(value_type&& obj);

template<
class P>
pair<
iterator, bool>
insert(P&& obj);

iterator
insert(const_iterator hint, const value_type& obj);
iterator
insert(const_iterator hint, value_type&& obj);

template<
class P>
iterator
insert(const_iterator hint, P&& obj);

template<
class InputIterator>
void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);

node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator
insert(const_iterator hint, node_type&& nh);

template<
class... Args>
pair<
iterator, bool>
try_emplace(const key_type& k, Args&&... args);

template<
class... Args>
pair<
iterator, bool>
try_emplace(key_type&& k, Args&&... args);

template<
class... Args>
iterator
try_emplace(const_iterator hint, const key_type& k, Args&&... args);

template<
class Arg>
iterator
try_emplace(const_iterator hint, key_type&& k, Arg&& obj);

template<
class M>
pair<
iterator, bool>
insert_or_assign(const key_type& k, M&& obj);

template<
class M>
pair<
iterator, bool>
insert_or_assign(key_type&& k, M&& obj);

template<
class M>
iterator
insert_or_assign(const_iterator hint, const key_type& k, M&& obj);

template<
class M>
iterator
insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);

void
swap(unordered_map&);

noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_swappable_v<Hash> &&
is_nothrow_swappable_v<Pred>);

void clear() noexcept;

template<
class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>& source);

template<
class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);

template<
class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);

template<
class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);
// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
iterator find(const key_type& k, size_t hash);
const_iterator find(const key_type& k, size_t hash) const;
template<class K>
iterator find(const K& k);
template<class K>
const_iterator find(const K& k) const;
template<class K>
iterator find(const K& k, size_t hash);
template<class K>
const_iterator find(const K& k, size_t hash) const;
size_type count(const key_type& k) const;
size_type count(const key_type& k, size_t hash) const;
template<class K>
size_type count(const K& k) const;
template<class K>
size_type count(const K& k, size_t hash) const;
bool contains(const key_type& k) const;
bool contains(const key_type& k, size_t hash) const;
template<class K>
bool contains(const K& k) const;
template<class K>
bool contains(const K& k, size_t hash) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k, size_t hash);
pair<const_iterator, const_iterator> equal_range(const key_type& k, size_t hash) 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;
template<class K>
pair<iterator, iterator> equal_range(const K& k, size_t hash);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& k, size_t hash) const;

// element access
mapped_type& operator[] (const key_type& k);
mapped_type& operator[] (key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);

};

template<class InputIterator,
    class Hash = hash<iter-key-type<InputIterator>>>,
    class Pred = equal_to<iter-key-type<InputIterator>>>,
    class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
unordered_map(InputIterator, InputIterator, typename see below::size_type = see below,
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash, Pred,
    Allocator>;

template<class Key, class T, class Hash = hash<Key>,
    class Pred = equal_to<Key>, class Allocator = allocator<pair<const Key, T>>>
unordered_map(initializer_list<pair<Key, T>>, typename see below::size_type = see below,
    Hash = Hash(),
    Pred = Pred(), Allocator = Allocator())
    -> unordered_map<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_map(InputIterator, InputIterator, typename see below::size_type, Allocator)
    -> unordered_map<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, hash<
        iter-key-type<InputIterator>>, equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_map(initializer_list<pair<Key, T>>, typename see below::size_type, Allocator)
    -> unordered_map<Key, T, hash<Key>, equal_to<Key>, Allocator>;

// swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
    unordered_map<Key, T, Hash, Pred, Alloc>& y)
    noexcept(noexcept(x.swap(y)));
}

A size_type parameter type in an unordered_map deduction guide refers to the size_type member type
of the type deduced by the deduction guide.

22.5.4.2 Constructors

unordered_map() : unordered_map(size_type(see below)) {}
explicit unordered_map(size_type n,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

1 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());

3 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);
```

1 Effects: Equivalent to: return try_emplace(k).first->second;

```
mapped_type& operator[](key_type&& k);
```

2 Effects: Equivalent to: return try_emplace(move(k)).first->second;

```
mapped_type& at(const key_type& k);
```

3 Returns: A reference to x.second, where x is the (unique) element whose key is equivalent to k.

```
const mapped_type& at(const key_type& k) const;
```

4 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);
```

1 Constraints: is_constructible_v<value_type, P&&> is true.

2 Effects: Equivalent to: return emplace(std::forward<P>(obj));

```
template<class P>
iterator insert(const_iterator hint, P&& obj);
```

3 Constraints: is_constructible_v<value_type, P&&> is true.

4 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);
```

5 Expects: 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);

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.

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.

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 [unord.map.erase]

```cpp
template<class K, class T, class H, class P, class A, class Predicate>
void erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);```

Effects: Equivalent to:

§ 22.5.4.5
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}

22.5.5 Class template unordered_multimap

22.5.5.1 Overview

1 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.

2 An unordered_multimap satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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>.

3 This subclause 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:
    // types
    using key_type = Key;
    using mapped_type = T;
    using value_type = pair<const Key, T>;
    using hasher = Hash;
    using key_equal = see 22.2.7;
    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(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, size_type n, const allocator_type& a)
    : unordered_multimap(il, n, hf, 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(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
    : unordered_multimap(f, l, n, hf, key_equal(), a) { }

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> il, size_type n, const allocator_type& a)
    : unordered_multimap(il, n, hasher(), key_equal(), a) { }

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(const_iterator hint, const value_type& 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);

// 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;
iterator find(const key_type& k, size_t hash);
const_iterator find(const key_type& k, size_t hash) const;
template<class K>
    iterator find(const K& k);
    template<class K>
    const_iterator find(const K& k) const;
    template<class K>
    iterator find(const K& k, size_t hash);
    template<class K>
    const_iterator find(const K& k, size_t hash) const;
size_type count(const key_type& k) const;
size_type count(const key_type& k, size_t hash) const;
    template<class K>
    size_type count(const K& k) const;
    template<class K>
    size_type count(const K& k, size_t hash) const;
bool contains(const key_type& k) const;
bool contains(const key_type& k, size_t hash) const;
    template<class K>
    bool contains(const K& k) const;
    template<class K>
    bool contains(const K& k, size_t hash) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k, size_t hash);
pair<const_iterator, const_iterator> equal_range(const key_type& k, size_t hash) 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;
    template<class K>
    pair<iterator, iterator> equal_range(const K& k, size_t hash);
    template<class K>
    pair<const_iterator, const_iterator> equal_range(const K& k, size_t hash) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);

};

template<class InputIterator,
class Hash = hash<iter-key-type<InputIterator>>,
class Pred = equal_to<iter-key-type<InputIterator>>,
class Allocator = allocator<iter-to-alloc-type<InputIterator>>>
unordered_multimap(InputIterator, InputIterator,
typename see below::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
Hash, Pred, Allocator>;

template<class Key, class T, class Hash = hash<Key>,
class Pred = equal_to<Key>, class Allocator = allocator<pair<const Key, T>>>
unordered_multimap(initializer_list<pair<Key, T>>,
typename see below::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_multimap<Key, T, Hash, Pred, Allocator>;

template<class InputIterator, class Allocator>
unordered_multimap(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
hash<iter-key-type<InputIterator>>,
equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Allocator>
unordered_multimap(InputIterator, InputIterator, Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>,
hash<iter-key-type<InputIterator>>,
equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multimap(InputIterator, InputIterator, typename see below::size_type, Hash,
Allocator)
-> unordered_multimap<iter-key-type<InputIterator>, iter-mapped-type<InputIterator>, Hash,
equal_to<iter-key-type<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(initializer_list<pair<Key, T>>, typename see below::size_type,
Allocator)
-> unordered_multimap<Key, T, hash<Key>>, equal_to<Key>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(initializer_list<pair<Key, T>>, Allocator)
-> unordered_multimap<Key, T, hash<Key>>, equal_to<Key>, Allocator>;
template<class Key, class T, class Hash, class Allocator>
unordered_multimap(initializer_list<pair<Key, T>>, typename
see below::size_type,
Hash, Allocator)
-> unordered_multimap<Key, T, Hash, equal_to<Key>, Allocator>;

// 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)));

4 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 [unord.multimap.cnstr]
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());
1
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());
3
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 [unord.multimap.modifiers]
template<class P>
iterator insert(P&& obj);
1 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);
3 Constraints: is_constructible_v<value_type, P&&> is true.
4 Effects: Equivalent to: return emplace_hint(hint, std::forward<P>(obj));

22.5.5.4 Erasure [unord.multimap.erasure]
template<class K, class T, class H, class P, class A, class Predicate>
void erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);
1 Effects: Equivalent to:
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}  

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 satisfies all of the requirements of a container, of an unordered associative container, and
of an allocator-aware container (Table 65). 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.

This subclause only describes operations on unordered_set that are not described in one of the requirement
tables, or for which there is additional semantic information.

namespace std {
    template<class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Allocator = allocator<Key>>
    class unordered_set {
        public:
            // types
            using key_type = Key;
            using value_type = Key;
            using hasher = Hash;
            using key_equal = see 22.2.7;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 22.2
            using difference_type = implementation-defined; // see 22.2
            using iterator = implementation-defined; // see 22.2
            using const_iterator = implementation-defined; // see 22.2
            using local_iterator = implementation-defined; // see 22.2
            using const_local_iterator = implementation-defined; // see 22.2
            using node_type = unspecified;
            using insert_return_type = insert-return-type<iterator, node_type>;

            // 22.5.6.2, construct/copy/destroy
            unordered_set();
            explicit unordered_set(size_type n,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
            template<class InputIterator>
            unordered_set(InputIterator f, InputIterator l,
                      size_type n = see below,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
            unordered_set(const unordered_set&);
            unordered_set(unordered_set&&);
explicit unordered_set(const Allocator&);
unordered_set(const unordered_set&, const Allocator&);
unordered_set(unordered_set&&, const Allocator&);
unordered_set(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
unordered_set(size_type n, const allocator_type& a)
    : unordered_set(n, hasher(), key_equal(), a) { }
unordered_set(size_type n, const hasher& hf, const allocator_type& a)
    : unordered_set(n, hf, key_equal(), a) { }
template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, size_type n, const allocator_type& a)
    : unordered_set(f, l, n, hasher(), key_equal(), a) { }
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, size_type n, const hasher& hf,
        const allocator_type& 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>);
allocate 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;

// 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;
iterator find(const key_type& k, size_t hash);
const_iterator find(const key_type& k, size_t hash) const;
template<class K>
iterator find(const K& k);
template<class K>
const_iterator find(const K& k) const;
template<class K>
iterator find(const K& k, size_t hash);
template<class K>
const_iterator find(const K& k, size_t hash) const;
size_type count(const key_type& k) const;
size_type count(const key_type& k, size_t hash) const;
template<class K>
size_type count(const K& k) const;
template<class K>
size_type count(const K& k, size_t hash) const;
bool contains(const key_type& k) const;
bool contains(const key_type& k, size_t hash) const;
template<class K>
bool contains(const K& k) const;
template<class K>
bool contains(const K& k, size_t hash) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k, size_t hash);
pair<const_iterator, const_iterator> equal_range(const key_type& k, size_t hash) const;
template<class K>
pair<iterator, iterator> equal_range(const K& k);
pair<const_iterator, const_iterator> equal_range(const K& k) const;
template<class K>
pair<iterator, iterator> equal_range(const K& k, size_t hash);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& k, size_t hash);

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template<class InputIterator,
class Hash = hash<iter-value-type<InputIterator>>,
class Pred = equal_to<iter-value-type<InputIterator>>,
class Allocator = allocator<iter-value-type<InputIterator>>>
unordered_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>;

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
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)));
}

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

[unord.set.erosure]

template<class K, class H, class P, class A, class Predicate>
void erase_if(unordered_set<K, H, P, A>& c, Predicate pred);

Effects: Equivalent to:
for (auto i = c.begin(), last = c.end(); i != last; ) {
    if (pred(*i)) {
        i = c.erase(i);
    } else {
        ++i;
    }
}

22.5.7 Class template unordered_multiset

[unord.multiset]

22.5.7.1 Overview

[unord.multiset.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 satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 65). 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.

This subclause only describes operations on unordered_multiset that are not described in one of the requirement tables, or for which there is additional semantic information.

namespace std {
    template<class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Allocator = allocator<Key>>
    class unordered_multiset {

§ 22.5.7.1 859
public:
   // types
   using key_type = Key;
   using value_type = Key;
   using hasher = Hash;
   using key_equal = see 22.2.7;
   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) { }
   template<class InputIterator>
   unordered_multiset(InputIterator f, InputIterator l, size_type n, const hasher& hf,
      const allocator_type& a)
      : unordered_multiset(f, l, n, hf, key_equal(), a) { }
   unordered_multiset(initializer_list<value_type> il, size_type n, const allocator_type& a)
      : unordered_multiset(il, n, hasher(), key_equal(), a) { }
   unordered_multiset(initializer_list<value_type> il, size_type n, const hasher& hf,
      const allocator_type& a)
      : unordered_multiset(il, n, hf, key_equal(), a) { }
   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);
node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);
iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void swap(unordered_multiset&)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_swappable_v<Hash> &&
    is_nothrow_swappable_v<Pred>);
void clear() noexcept;

template<class H2, class P2>
    void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
    void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);
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;
iterator find(const key_type& k, size_t hash);
const_iterator find(const key_type& k, size_t hash) const;
template<class K>
    iterator find(const K& k);
template<class K>
    const_iterator find(const K& k) const;
template<class K>
    iterator find(const K& k, size_t hash);

template<class K>
    const_iterator find(const K& k, size_t hash) const;

size_type count(const key_type& k) const;

size_type count(const key_type& k, size_t hash) const;

template<class K>
    size_type count(const K& k) const;

template<class K>
    size_type count(const K& k, size_t hash) const;

bool contains(const key_type& k) const;

bool contains(const key_type& k, size_t hash) const;

template<class K>
    bool contains(const K& k) const;

template<class K>
    bool contains(const K& k, size_t hash) const;

template<class K>
    pair<iterator, iterator> equal_range(const key_type& k);

pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

pair<iterator, iterator> equal_range(const key_type& k, size_t hash);

pair<const_iterator, const_iterator> equal_range(const key_type& k, size_t hash) 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;

template<class K>
    pair<iterator, iterator> equal_range(const K& k, size_t hash);

template<class K>
    pair<const_iterator, const_iterator> equal_range(const K& k, size_t hash) const;

// bucket interface
size_type bucket_count() const noexcept;

size_type max_bucket_count() const noexcept;

size_type bucket(size_type n) const;

local_iterator begin(size_type n);

const_local_iterator begin(size_type n) const;

local_iterator end(size_type n);

const_local_iterator end(size_type n) const;

const_local_iterator cbegin(size_type n) const;

const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;

float max_load_factor() const noexcept;

void max_load_factor(float z);

void reserve(size_type n);

};

template<class InputIterator,
    class Hash = hash<iter_value_type<InputIterator>>,
    class Pred = equal_to<iter_value_type<InputIterator>>,
    class Allocator = allocator<iter_value_type<InputIterator>>> unordered_multiset(InputIterator, InputIterator,
        see below::size_type = see below,
        Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multiset<iter_value_type<InputIterator>>,
        Hash, Pred, Allocator>;

template<class T, class Hash = hash<T>,
    class Pred = equal_to<T>, class Allocator = allocator<T>>
unordered_multiset(initializer_list<T>, typename
    see below::size_type = see below,
        Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multiset<T, Hash, Pred, Allocator>;
template<class InputIterator, class Allocator>
unordered_multiset(InputIterator, InputIterator, typename see below::size_type, Allocator)
-> unordered_multiset<iter-value-type<InputIterator>,
    hash<iter-value-type<InputIterator>>,
    equal_to<iter-value-type<InputIterator>>,
    Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multiset(InputIterator, InputIterator, typename see below::size_type,
    Hash, Allocator)
-> unordered_multiset<iter-value-type<InputIterator>, Hash,
    equal_to<iter-value-type<InputIterator>>,
    Allocator>;

template<class T, class Allocator>
unordered_multiset(initializer_list<T>, typename see below::size_type, Allocator)
-> unordered_multiset<T, hash<T>, equal_to<T>, Allocator>;

template<class T, class Hash, class Allocator>
unordered_multiset(initializer_list<T>, typename see below::size_type, Hash, Allocator)
-> unordered_multiset<T, Hash, equal_to<T>, Allocator>;

// swap
template<class Key, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
    unordered_multiset<Key, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)))
{

}  

A size_type parameter type in an unordered_multiset deduction guide refers to the size_type member
type of the type deduced by the deduction guide.

22.5.7.2 Constructors [unord.multiset.cnstr]

unordered_multiset() : unordered_multiset(size_type(see below)) {}  
explicit unordered_multiset(size_type n,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multiset using the specified hash function, key equality
predicate, and allocator, and using at least n buckets. For the default constructor, the number of
buckets is implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

unordered_multiset(initializer_list<value_type> il,
    size_type n = see below,
    const hasher& hf = hasher(),
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multiset using the specified hash function, key equality
predicate, and allocator, and using at least n buckets. If n is not provided, the number of buckets is
implementation-defined. Then inserts elements from the range [f, l) for the first form, or from the
range [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.
22.5.7.3 Erasure

```cpp
template<class K, class H, class P, class A, class Predicate>
void erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);
```

1. **Effects:** Equivalent to:
   ```cpp
   for (auto i = c.begin(), last = c.end(); i != last; ) {
     if (pred(*i)) {
       i = c.erase(i);
     } else {
       ++i;
     }
   }
   ```

22.6 Container adaptors

22.6.1 In general

The headers `<queue>` and `<stack>` define the container adaptors `queue`, `priority_queue`, and `stack`.

The container adaptors each take a `Container` template parameter, and each constructor takes a `Container` reference argument. This container is copied into the `Container` member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor’s constructor. Otherwise, normal copy or move construction is used for the container argument. The first template parameter `T` of the container adaptors shall denote the same type as `Container::value_type`.

For container adaptors, no `swap` function throws an exception unless that exception is thrown by the swap of the adaptor’s `Container` or `Compare` object (if any).

A deduction guide for a container adaptor shall not participate in overload resolution if any of the following are true:

1. **(4.1)** It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
2. **(4.2)** It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.
3. **(4.3)** It has a `Container` template parameter and a type that qualifies as an allocator is deduced for that parameter.
4. **(4.4)** It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
5. **(4.5)** It has both `Container` and `Allocator` template parameters, and `uses_allocator_v<Container, Allocator>` is false.

22.6.2 Header `<queue>` synopsis

```cpp
#include <initializer_list>

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>
    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>;

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

```
#include <initializer_list>

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>
        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.

```
template<class Alloc> explicit queue(const Alloc&);
template<class Alloc> queue(const Container& cont, const Alloc&);
template<class Alloc> queue(const Alloc& a);

Effects: Initializes c with cont.

template<class Alloc> queue(Container&& cont);
template<class Alloc> queue(const Container& cont, const Alloc& a);

Effects: Initializes c with cont.

22.6.4.2 Constructors

explicit queue(const Container& cont);

Effects: Initializes c with cont.

explicit queue(Container&& cont);

Effects: Initializes c with std::move(cont).

22.6.4.3 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 queue(const Alloc& a);

Effects: Initializes c with a.

template<class Alloc> queue(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> queue(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> queue(const queue& q, const Alloc& a);

Effects: Initializes c with q as the first argument and a as the second argument.

template<class Alloc> queue(queue&& q, const Alloc& a);

Effects: Initializes c with std::move(q) as the first argument and a as the second argument.
22.6.4.4 Operators

```cpp
template<class T, class Container>
bool operator== (const queue<T, Container>& x, const queue<T, Container>& y);
1
Returns: x.c == y.c.
```

```cpp
template<class T, class Container>
bool operator!= (const queue<T, Container>& x, const queue<T, Container>& y);
2
Returns: x.c != y.c.
```

```cpp
template<class T, class Container>
bool operator< (const queue<T, Container>& x, const queue<T, Container>& y);
3
Returns: x.c < y.c.
```

```cpp
template<class T, class Container>
bool operator> (const queue<T, Container>& x, const queue<T, Container>& y);
4
Returns: x.c > y.c.
```

```cpp
template<class T, class Container>
bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);
5
Returns: x.c <= y.c.
```

```cpp
template<class T, class Container>
bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);
6
Returns: x.c >= y.c.
```

22.6.4.5 Specialized algorithms

```cpp
template<class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));
1
Constraints: is_swappable_v<Container> is true.
```

§ 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 (25.7).

```cpp
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& y);
priority_queue(const Compare& x, Container&& y);

1 Expects: x defines a strict weak ordering (25.7).

2 Effects: Initializes comp with x and c with y (copy constructing or move constructing as appropriate); calls make_heap(c.begin(), c.end(), comp).

template<class InputIterator>
priority_queue(InputIterator first, InputIterator last, const Compare& x, const Container& y);
template<class InputIterator>
    priority_queue(InputIterator first, InputIterator last, const Compare& x = Compare(),
                    Container&& y = Container());

    **Expects:** x defines a strict weak ordering (25.7).

    **Effects:** Initializes comp with x and c with y (copy constructing or move constructing as appropriate);
                 calls c.insert(c.end(), first, last); and finally calls make_heap(c.begin(), c.end(), comp).

22.6.5.3 Constructors with allocators [priqueue.cons.alloc]

If uses_allocator_v<container_type, Alloc> is false the constructors in this subclause shall not participate in overload resolution.

    template<class Alloc> explicit priority_queue(const Alloc& a);

    **Effects:** Initializes c with a and value-initializes comp.

    template<class Alloc> priority_queue(const Compare& compare, const Alloc& a);

    **Effects:** Initializes c with a and initializes comp with compare.

    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 [priqueue.members]

    void push(const value_type& x);

    **Effects:** As if by:
                 c.push_back(x);
                 push_heap(c.begin(), c.end(), comp);

    void push(value_type&& x);

    **Effects:** As if by:
                 c.push_back(std::move(x));
                 push_heap(c.begin(), c.end(), comp);

    template<class... Args> void emplace(Args&&... args)

    **Effects:** As if by:
                 c.emplace_back(std::forward<Args>(args)...);
                 push_heap(c.begin(), c.end(), comp);

    void pop();

    **Effects:** As if by:
                 pop_heap(c.begin(), c.end(), comp);
                 c.pop_back();
22.6.5.5 Specialized algorithms

```cpp
template<class T, class Container, class Compare>
void swap(priority_queue<T, Container, Compare>& x,
          priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));
```

Constraints: `is_swappable_v<Container>` is true and `is_swappable_v<Compare>` is true.

Effects: As if by `x.swap(y)`.

22.6.6 Class template stack

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

```cpp
namespace std {
    template<class T, class Container = deque<T>>
    class stack {
        public:
            using value_type = typename Container::value_type;
            using reference = typename Container::reference;
            using const_reference = typename Container::const_reference;
            using size_type = typename Container::size_type;
            using container_type = Container;

        protected:
            Container c;

        public:
            stack() : stack(Container()) {} // 870
            explicit stack(const Container&);
            explicit stack(Container&&);
            template<class Alloc> explicit stack(const Alloc&);
            template<class Alloc> stack(const Container&, const Alloc&);
            template<class Alloc> stack(Container&&, const Alloc&);
            template<class Alloc> stack(const stack&, const Alloc&);
            template<class Alloc> stack(stack&&, const Alloc&);
            [[nodiscard]] bool empty() const { return c.empty(); }
            size_type size() const { return c.size(); }
            reference top() { return c.back(); }
            const_reference top() const { return c.back(); }
            void push(const value_type& x) { c.push_back(x); }
            void push(value_type&& x) { c.push_back(std::move(x)); }
            template<class... Args>
            decltype(auto) emplace(Args&&... args) { return c.emplace_back(std::forward<Args>(args)...); }
            void pop() { c.pop_back(); }
            void swap(stack& s) noexcept(is_nothrow_swappable_v<Container>) { using std::swap; swap(c, s.c); }

    };

    template<class Container>
    stack(Container) -> stack<typename Container::value_type, Container>;

    template<class Container, class Allocator>
    stack(Container, Allocator) -> stack<typename Container::value_type, Container>;

    template<class T, class Container, class Alloc>
    struct uses_allocator<stack<T, Container>, Alloc>
        : uses_allocator<Container, Alloc>::type { }; // 870

}
```
22.6.6.2 Constructors

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

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(const 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.

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 [views]
22.7.1 General [views.general]

1 The header `<span>` defines the view span.

22.7.2 Header `<span>` synopsis [span.syn]

```cpp
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;

    // 22.7.3.7, 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;

    // 22.7.3.8, tuple interface
    template<class T> class tuple_size;
    template<size_t I, class T> class tuple_element;
    template<class ElementType, size_t Extent>
    struct tuple_size<span<ElementType, Extent>>;
    template<class ElementType>
    struct tuple_size<span<ElementType, dynamic_extent>>;
    // not defined
    template<size_t I, class ElementType, size_t Extent>
    constexpr ElementType& get(span<ElementType, Extent>) noexcept;
}
```

22.7.3 Class template span [views.span]

22.7.3.1 Overview [span.overview]

1 A span is a view over a contiguous sequence of objects, the storage of which is owned by some other object.

2 The iterator types `span::iterator` and `span::const_iterator` model `ContiguousIterator` (23.3.4.14), meet the `Cpp17RandomAccessIterator` requirements (23.3.5.6), and meet the requirements for `constexpr` iterators (23.3.1). All requirements on container iterators (22.2) apply to `span::iterator` and `span::const_iterator` as well.

3 All member functions of `span` have constant time complexity.

```cpp
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 index_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;
            using const_iterator = implementation-defined;
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    }
```
static constexpr index_type extent = Extent;

// 22.7.3.2, constructors, copy, and assignment
constexpr span() noexcept;
constexpr span(pointer ptr, index_type count);
constexpr span(pointer first, pointer last);
template<size_t N>
  constexpr span(element_type (&arr)[N]) noexcept;
template<size_t N>
  constexpr span(array<value_type, N>& arr) noexcept;
template<size_t N>
  constexpr span(const array<value_type, N>& arr) noexcept;
template<class Container>
  constexpr span(Container& cont);
template<class Container>
  constexpr span(const Container& cont);
constexpr span(const span& other) noexcept = default;
template<class OtherElementType, size_t OtherExtent>
  constexpr span(const span<OtherElementType, OtherExtent>& s) noexcept;
~span() noexcept = default;
constexpr span& operator=(const span& other) noexcept = default;

// 22.7.3.3, subviews
template<size_t Count>
  constexpr span<element_type, Count> first() const;
template<size_t Count>
  constexpr span<element_type, Count> last() const;
template<size_t Offset, size_t Count = dynamic_extent
  constexpr span<element_type, see below> subspan() const;
constexpr span<element_type, dynamic_extent> first(index_type count) const;
constexpr span<element_type, dynamic_extent> last(index_type count) const;
constexpr span<element_type, dynamic_extent> subspan(
  index_type offset, index_type count = dynamic_extent) const;

// 22.7.3.4, observers
constexpr index_type size() const noexcept;
constexpr index_type size_bytes() const noexcept;
[[nodiscard]] constexpr bool empty() const noexcept;

// 22.7.3.5, element access
constexpr reference operator[](index_type idx) const;
constexpr reference front() const;
constexpr reference back() const;
constexpr pointer data() const noexcept;

// 22.7.3.6, iterator support
constexpr iterator begin() const noexcept;
constexpr iterator end() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

friend constexpr iterator begin(span s) noexcept { return s.begin(); }
friend constexpr iterator end(span s) noexcept { return s.end(); }

private:
  pointer data_; // exposition only
  index_type size_; // exposition only
template<class T, size_t N>
span(T (&)[N]) -> span<T, N>;
template<class T, size_t N>
span(array<T, N>&) -> span<T, N>;
template<class T, size_t N>
span(const array<T, N>&) -> span<const T, N>;
template<class Container>
span(Container&) -> span<typename Container::value_type>;
template<class Container>
span(const Container&) -> span<const typename Container::value_type>;

ElementType
is required to be a complete object type that is not an abstract class type.

22.7.3.2 Constructors, copy, and assignment

constexpr span() noexcept;

Constraints: Extent \leq 0 is true.

Ensures: size() == 0 && data() == nullptr.

constexpr span(pointer ptr, index_type count);

Expects: [ptr, ptr + count) is a valid range. If extent is not equal to dynamic_extent, then count
equal to extent.

Effects: Constructs a span that is a view over the range [ptr, ptr + count).

Ensures: size() == count && data() == ptr.

Throws: Nothing.

constexpr span(pointer first, pointer last);

Expects: [first, last) is a valid range. If extent is not equal to dynamic_extent, then last -
first is equal to extent.

Effects: Constructs a span that is a view over the range [first, last).

Ensures: size() == last - first && data() == first.

Throws: Nothing.

template<size_t N> constexpr span(element_type (&arr)[N]) noexcept;

Constraints:

(11.1) extent == dynamic_extent || N == extent is true, and
(11.2) remove_pointer_t<decltype(data(arr))>(*)[] is convertible to ElementType(*[]).

Effects: Constructs a span that is a view over the supplied array.

Ensures: size() == N && data() == data(arr).

template<class Container> constexpr span(Container& cont);

Constraints:

(14.1) extent == dynamic_extent is true,
(14.2) Container is not a specialization of span,
(14.3) Container is not a specialization of array,
(14.4) is_array_v<Container> is false,
(14.5) data(cont) and size(cont) are both well-formed, and
(14.6) remove_pointer_t<decltype(data(cont))>(*)[] is convertible to ElementType(*[]).

§ 22.7.3.2
Expects: \([\text{data}(\text{cont}), \text{data}(\text{cont}) + \text{size}(\text{cont})]\) is a valid range.

Effects: Constructs a span that is a view over the range \([\text{data}(\text{cont}), \text{data}(\text{cont}) + \text{size}(\text{cont})]\).

Ensures: \(\text{size}() = \text{size}(\text{cont}) \land \text{&data}() = \text{data}(\text{cont})\).

Throws: What and when \(\text{data}(\text{cont})\) and \(\text{size}(\text{cont})\) throw.

constexpr span(const span& other) noexcept = default;

Ensures: \(\text{other.size}() = \text{size}() \land \text{&other.data}() = \text{data}()\).

template<class OtherElementType, size_t OtherExtent>
constexpr span(const span<OtherElementType, OtherExtent>& s) noexcept;

Constraints:

(20.1) \(\text{Extent} = \text{dynamic_extent} \land \text{Extent} = \text{OtherExtent}\) is true, and

(20.2) \(\text{OtherElementType(*)[]}\) is convertible to \(\text{ElementType(*)[]}\).

Effects: Constructs a span that is a view over the range \([s.\text{data}(), s.\text{data}() + s.\text{size}()]\).

Ensures: \(\text{size}() = s.\text{size}() \land \text{&data}() = s.\text{data}()\).

constexpr span& operator=(const span& other) noexcept = default;

Ensures: \(\text{size}() = \text{other.size}() \land \text{&data}() = \text{other.data}()\).

22.7.3.3 Subviews

template<size_t Count> constexpr span<element_type, Count> first() const;

Expects: \(\text{Count} \leq \text{size}()\) is true.

Effects: Equivalent to: return \{\text{data}(), Count\};

template<size_t Count> constexpr span<element_type, Count> last() const;

Expects: \(\text{Count} \leq \text{size}()\) is true.

Effects: Equivalent to: return \{\text{data}() + (\text{size}() - Count), Count\};

template<size_t Offset, size_t Count = dynamic_extent>
constexpr span<element_type, see below> subspan() const;

Expects:

Offset \(\leq \text{size}() \land (\text{Count} = \text{dynamic_extent} \lor \text{Offset} + \text{Count} \leq \text{size}())\) is true.

Effects: Equivalent to:

return span<element_type, see below>(
    \text{data}() + Offset, Count = dynamic_extent ? Count : \text{size}() - Offset);

Remarks: The second template argument of the returned span type is:

\(\text{Count} \neq \text{dynamic_extent} \land \text{Count}
\)

\(\lnot \text{Extent} \neq \text{dynamic_extent} \land \text{Extent} - \text{Offset}
\)

\(\text{dynamic_extent}\)

constexpr span<element_type, dynamic_extent> first(index_type count) const;

Expects: \(\text{count} \leq \text{size}()\) is true.

Effects: Equivalent to: return \{\text{data}(), count\};

constexpr span<element_type, dynamic_extent> last(index_type count) const;

Expects: \(\text{count} \leq \text{size}()\) is true.

Effects: Equivalent to: return \{\text{data}() + (\text{size}() - count), count\};

constexpr span<element_type, dynamic_extent> subspan(
    index_type offset, index_type count = dynamic_extent) const;

Expects:
offset <= size() && (count == dynamic_extent || offset + count <= size()) is true.

**Effects:** Equivalent to:

\[
\text{return } \{\text{data() }+ \text{offset, count }= \text{dynamic_extent }? \text{size() }- \text{offset : count}\};
\]

### 22.7.3.4 Observers

```cpp
constexpr index_type size() const noexcept;
```

**Effects:** Equivalent to: \text{return size_;}

```cpp
constexpr index_type size_bytes() const noexcept;
```

**Effects:** Equivalent to: \text{return size() }\ast \text{sizeof(element_type)};

```cpp
[[nodiscard]] constexpr bool empty() const noexcept;
```

**Effects:** Equivalent to: \text{return size() }== 0;

### 22.7.3.5 Element access

```cpp
constexpr reference operator[](index_type idx) const;
```

**Expects:** \text{idx }< \text{size()} is true.

**Effects:** Equivalent to: \text{return }*(\text{data() }+ \text{idx});

```cpp
constexpr reference front() const;
```

**Expects:** \text{empty()} is false.

**Effects:** Equivalent to: \text{return }*\text{data();}

```cpp
constexpr reference back() const;
```

**Expects:** \text{empty()} is false.

**Effects:** Equivalent to: \text{return }*(\text{data() }+ (\text{size() }- 1));

```cpp
constexpr pointer data() const noexcept;
```

**Effects:** Equivalent to: \text{return data_;}

### 22.7.3.6 Iterator support

```cpp
constexpr iterator begin() const noexcept;
```

**Returns:** An iterator referring to the first element in the span. If \text{empty()} is \text{true}, then it returns the same value as \text{end();}

```cpp
constexpr iterator end() const noexcept;
```

**Returns:** An iterator which is the past-the-end value.

```cpp
constexpr reverse_iterator rbegin() const noexcept;
```

**Effects:** Equivalent to: \text{return }\text{reverse_iterator(end());}

```cpp
constexpr reverse_iterator rend() const noexcept;
```

**Returns:** Equivalent to: \text{return }\text{reverse_iterator(begin());}

```cpp
constexpr const_iterator cbegin() const noexcept;
```

**Returns:** An iterator referring to the first element in the span. If \text{empty()} is \text{true}, then it returns the same value as \text{cend();}

```cpp
constexpr const_iterator cend() const noexcept;
```

**Returns:** A constant iterator which is the past-the-end value.

```cpp
constexpr const_reverse_iterator crbegin() const noexcept;
```

**Effects:** Equivalent to: \text{return }\text{const_reverse_iterator(cend());}

```cpp
constexpr const_reverse_iterator crend() const noexcept;
```

**Effects:** Equivalent to: \text{return }\text{const_reverse_iterator(cend());}
constexpr const_reverse_iterator crend() const noexcept;

Effects: Equivalent to: return const_reverse_iterator(cbegin());

22.7.3.7 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 {reinterpret_cast<const byte*>(s.data()), s.size_bytes();

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 {reinterpret_cast< byte*>(s.data()), s.size_bytes();

22.7.3.8 Tuple interface

template<class ElementType, size_t Extent>
struct tuple_size< span<ElementType, Extent> >
: integral_constant<size_t, Extent> { };

tuple_element<I, span<ElementType, Extent> >::type

Mandates: Extent != dynamic_extent && I < Extent is true.

Value: The type ElementType.

template<size_t I, class ElementType, size_t Extent>
constexpr ElementType& get(span<ElementType, Extent> s) noexcept;

Mandates: Extent != dynamic_extent && I < Extent is true.

Returns: A reference to the I^{th} element of s, where indexing is zero-based.
23 Iterators library

23.1 General

1 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).

2 The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 71.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.3</td>
<td>&lt;iterator&gt;</td>
</tr>
<tr>
<td>23.4</td>
<td></td>
</tr>
<tr>
<td>23.5</td>
<td></td>
</tr>
<tr>
<td>23.6</td>
<td></td>
</tr>
<tr>
<td>23.7</td>
<td></td>
</tr>
</tbody>
</table>

23.2 Header <iterator> synopsis

```cpp
#include <concepts>
namespace std {
    template<class T> using with-reference = T&; // exposition only
    template<class T> concept can-reference // exposition only
        = requires { typename with-reference<T>; }
        { *t } -> can-reference; // not required to be equality-preserving
    template<class T> concept dereferenceable // exposition only
        = requires(T& t) {
            { *t } -> can-reference;
        };

    // 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, readable traits
    template<class> struct 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> struct iterator_traits<T*>;

    template<dereferenceable 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;

            // 23.3.3.2, ranges::iter_swap
            inline constexpr unspecified iter_swap = unspecified;
        }
    }
```

§ 23.2 878
template<\ dereferenceable\ T>  
   requires requires(T& t) {  
      { ranges::iter_move(t) } \to\ can-reference;  
   }  
using iter_rvalue_reference_t  
   = decltype(ranges::iter_move(declval<T&>()));  

// 23.3.4, iterator concepts  
// 23.3.4.2, concept Readable  
template<class In>  
   concept Readable = see below;  

template<Readable T>  
   using iter_common_reference_t  
      = common_reference_t<iter_reference_t<T>, iter_value_t<T>&>;  

// 23.3.4.3, concept Writable  
// 23.3.4.4, concept WeaklyIncrementable  
// 23.3.4.5, concept Incrementable  
// 23.3.4.6, concept Iterator  
// 23.3.4.7, concept Sentinel  
// 23.3.4.8, concept SizedSentinel  
// 23.3.4.9, concept InputIterator  
// 23.3.4.10, concept OutputIterator  
// 23.3.4.11, concept ForwardIterator  
// 23.3.4.12, concept BidirectionalIterator  
// 23.3.4.13, concept RandomAccessIterator
template<concept ContiguousIterator = see below;>

template<concept IndirectUnaryInvocable = see below;>

template<concept IndirectRegularUnaryInvocable = see below;>

template<concept IndirectUnaryPredicate = see below;>

template<concept IndirectRelation = see below;>

template<concept IndirectStrictWeakOrder = see below;>

template<concept IndirectlyMovable = see below;>

template<concept IndirectlyMovableStorable = see below;>

template<concept IndirectlyCopyable = see below;>

template<concept IndirectlyCopyableStorable = see below;>

template<concept IndirectlySwappable = see below;>

template<concept IndirectlyComparable = see below;>

template<concept Permutable = see below;>

template<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<Iterator I>
    constexpr void advance(I& i, iter_difference_t<I> n);
    template<Iterator I, Sentinel<I> S>
    constexpr void advance(I& i, S bound);
    template<Iterator I, Sentinel<I> S>
    constexpr iter_difference_t<I> advance(I& i, iter_difference_t<I> n, S bound);

    // 23.4.3.2, ranges::distance
    template<Iterator I, Sentinel<I> S>
    constexpr iter_difference_t<I> distance(I first, S last);
    template<Range R>
    constexpr iter_difference_t<iterator_t<R>> distance(R&& r);

    // 23.4.3.3, ranges::next
    template<Iterator I>
    constexpr I next(I x);
    template<Iterator I>
    constexpr I next(I x, iter_difference_t<I> n);
    template<Iterator I, Sentinel<I> S>
    constexpr I next(I x, S bound);
    template<Iterator I, Sentinel<I> S>
    constexpr I next(I x, iter_difference_t<I> n, S bound);

    // 23.4.3.4, ranges::prev
    template<BidirectionalIterator I>
    constexpr I prev(I x);
    template<BidirectionalIterator I>
    constexpr I prev(I x, iter_difference_t<I> n);
    template<BidirectionalIterator 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, 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 Iterator1, class Iterator2>
requires (!SizedSentinel<Iterator1, Iterator2>)
inline constexpr bool disable_sized_sentinel<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, iterator_t<Container> i);

// 23.5.3. move iterators and sentinels

template<class Iterator> class move_iterator;
template<class Iterator1, class Iterator2>
constexpr bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr auto operator-(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());

template<class Iterator1, class Iterator2>
constexpr move_iterator<Iterator1> operator+(typename move_iterator<Iterator1>::difference_type n, const move_iterator<Iterator1>& x);

template<class Iterator1, class Iterator2>
constexpr move_iterator<Iterator2> make_move_iterator(Iterator i);

template<Semiregular S> class move_sentinel;

// 23.5.4, common iterators
template<Iterator I, Sentinel<I> S>
requires (!Same<I, S>)
class common_iterator;

template<class I, class S>
struct incrementable_traits<common_iterator<I, S>>;

template<InputIterator I, class S>
struct iterator_traits<common_iterator<I, S>>;

// 23.5.5, default sentinels
struct default_sentinel_t;
inline constexpr default_sentinel_t default_sentinel{};

// 23.5.6, counted iterators
template<Iterator I> class counted_iterator;

template<class I>
struct incrementable_traits<counted_iterator<I>>;

template<InputIterator 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<T, charT = char, class traits = char_traits<charT>,
class Distance = ptrdiff_t>
class istream_iterator;
template<class T, class charT, class traits, class Distance>
bool operator==<const istream_iterator<T,charT,traits,Distance>& x, 
const istream_iterator<T,charT,traits,Distance>& y);

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);

template<class T, class charT = char, class traits = char_traits<charT>>
class ostream_iterator;

template<class T, class charT = char, class traits = char_traits<charT>>
class istreambuf_iterator;

// 23.7, range access
template<class C> constexpr auto begin(C& c) noexcept(noexcept(std::begin(c))) 
-> decltype(std::begin(c));

template<class C> constexpr auto begin(const C& c) noexcept(noexcept(std::begin(c))) 
-> decltype(std::begin(c));

template<class C> constexpr auto end(C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));

template<class C> constexpr auto end(const C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));

// 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 \( \ast i \), 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 writable to the iterator; for each such type \( T \), the expression \( \ast i = o \) is valid where \( o \) is a value of type \( T \). For every iterator type \( X \), there is a corresponding signed integer type called the difference type of the iterator.

2 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 72.

3 The six categories of iterators correspond to the iterator concepts InputIterator (23.3.4.9), OutputIterator (23.3.4.10), ForwardIterator (23.3.4.11), BidirectionalIterator (23.3.4.12) RandomAccessIterator (23.3.4.13), and ContiguousIterator (23.3.4.14), respectively. The generic term iterator refers to any type that models the Iterator concept (23.3.4.6).

4 Forward iterators satisfy all the requirements of input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified; Contiguous iterators also satisfy all the requirements of random access iterators and can be used whenever a random access iterator is specified.

5 Iterators that further satisfy the requirements of output iterators are called mutable iterators. Nonmutable iterators are referred to as constant iterators.

6 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]

7 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. These values are called past-the-end values. Values of an iterator \( i \) for which the expression \( \ast i \) 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 satisfy 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.

8 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.232

9 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 \( j == s \).

10 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.

232 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.

Table 72 — 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>

§ 23.3.1
A counted range \([i, n)\) is empty if \(n == 0\); otherwise, \([i, n)\) 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 \([i, n)\) is valid if and only if \(n == 0\); or \(n\) is positive, \(i\) is dereferenceable, and \([i + 1, --n)\) 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.233

Iterators are called constexpr iterators if all operations provided to meet iterator category requirements are constexpr functions, except for

(16.1) — a pseudo-destructor call (7.5.4.3), and

(16.2) — the construction of an iterator with a singular value.

[Note: For example, the types “pointer to int” and reverse_iterator<int*> are constexpr iterators. — end note]

### 23.3.2 Associated types

#### 23.3.2.1 Incrementable traits

To implement algorithms only in terms of incrementable types, it is often necessary to determine the difference type that corresponds to a particular incrementable type. Accordingly, it is required that if \(WI\) is the name of a type that models the WeaklyIncrementable concept (23.3.4.4), the type

\[\text{iter_difference_t}<\text{WI}>\]

be defined as the incrementable type’s difference type.

```cpp
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; }
    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;
}
```

233) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.
The type \texttt{iter\_difference\_t\<I\>} denotes

\begin{enumerate}
\item \texttt{incrementable\_traits\<I\>::\texttt{difference\_type}} if \texttt{iterator\_traits\<I\>} names a specialization generated from the primary template, and
\item \texttt{iterator\_traits\<I\>::\texttt{difference\_type}} otherwise.
\end{enumerate}

Users may specialize \texttt{incrementable\_traits} on program-defined types.

\subsection*{23.3.2.2 Readable traits \[\texttt{readable.traits}\]}

To implement algorithms only in terms of readable types, it is often necessary to determine the value type that corresponds to a particular readable type. Accordingly, it is required that if \texttt{R} is the name of a type that models the \texttt{Readable} concept (23.3.4.2), the type

\begin{verbatim}
iter\_value\_t\<R>
\end{verbatim}

be defined as the readable type's value type.

\begin{verbatim}
template<class> struct cond\_value\_type { }; // exposition only
template<class T>
  requires is\_object\_v<T>
  struct cond\_value\_type {
    using value\_type = remove\_cv\_t<T>;
  };

template<class> struct readable\_traits { };

template<class T>
  struct readable\_traits<T*> {
    : cond\_value\_type<T> { };

  template<class I>
    requires is\_array\_v<I>
    struct readable\_traits<I> {
      using value\_type = remove\_cv\_t<remove\_extent\_t<I>>;
    };

  template<class I>
    struct readable\_traits<const I> {
      : readable\_traits<I> { };

  template<class T>
    requires requires { typename T::value\_type; }
    struct readable\_traits<T> {
      : cond\_value\_type<typename T::value\_type> { };

    template<class T>
      requires requires { typename T::element\_type; }
      struct readable\_traits<T> {
        : cond\_value\_type<typename T::element\_type> { };

      template<class T> using iter\_value\_t = see below;
\end{verbatim}

The type \texttt{iter\_value\_t\<I\>} denotes

\begin{enumerate}
\item \texttt{readable\_traits\<I\>::\texttt{value\_type}} if \texttt{iterator\_traits\<I\>} names a specialization generated from the primary template, and
\item \texttt{iterator\_traits\<I\>::\texttt{value\_type}} otherwise.
\end{enumerate}

Class template \texttt{readable\_traits} may be specialized on program-defined types.

[\textit{Note:} Some legacy output iterators define a nested type named \texttt{value\_type} that is an alias for \texttt{void}. These types are not \texttt{Readable} and have no associated value types. \textit{- end note}]

[\textit{Note:} Smart pointers like \texttt{shared\_ptr<int>} are \texttt{Readable} and have an associated value type, but a smart pointer like \texttt{shared\_ptr<void>} is not \texttt{Readable} and has no associated value type. \textit{- end note}]

\section*{§ 23.3.2.2}
23.3.2.3 Iterator traits

To implement algorithms only in terms of iterators, it is sometimes necessary to determine the iterator category that corresponds to a particular iterator type. Accordingly, it is required that if \( I \) is the type of an iterator, the type

\[
\text{iterator_traits}<I>::\text{iterator_category}
\]

be defined as the iterator's iterator category. In addition, the types

\[
\text{iterator_traits}<I>::\text{pointer}
\]
\[
\text{iterator_traits}<I>::\text{reference}
\]

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

\[
\text{iterator_traits}<I>::\text{value_type}
\]
\[
\text{iterator_traits}<I>::\text{difference_type}
\]
\[
\text{iterator_traits}<I>::\text{reference}
\]

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<I&>;
        *i++ -> can-reference;
    };

template<class I>
concept cpp17-input-iterator =
    cpp17-iterator<I> && EqualityComparable<I> && requires(I i) {
        typename incrementable_traits<I>::difference_type;
        typename readable_traits<I>::value_type;
        typename common_reference_t<iter_reference_t<I>&&,
            typename readable_traits<I>::value_type&>;
        *i++;
        typename common_reference_t<decltype(*i++)&&,
            typename readable_traits<I>::value_type&>;
    requires SignedIntegral<typename incrementable_traits<I>::difference_type>;
};

template<class I>
concept cpp17-forward-iterator =
    cpp17-input-iterator<I> && Constructible<I> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    Same<remove_cvref_t<iter_reference_t<I>>, typename readable_traits<I>::value_type>
    && requires(I i) {
        i++ -> const I&;
        *i++ -> Same<iter_reference_t<I>>;
    };

template<class I>
concept cpp17-bidirectional-iterator =
    cpp17-forward-iterator<I> && requires(I i) {
        --i -> Same<I&>;
        i-- -> const I&;
        *i-- -> Same<iter_reference_t<I>>;
    };

template<class I>
concept cpp17-random-access-iterator =
    cpp17-bidirectional-iterator<I> && StrictTotallyOrdered<I> &&
```
The members of a specialization `iterator_traits<I>` generated from the `iterator_traits` primary template are computed as follows:

1. If `I` has valid (13.9.2) 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 = typename I::reference;
   ```

   If the `qualified-id` `I::pointer` is valid and denotes a type, then `iterator_traits<I>::pointer` names that type; otherwise, it names `void`.

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 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:

   (3.2.3.1) `random_access_iterator_tag` if `I` satisfies `cpp17-random-access-iterator`, or otherwise
   (3.2.3.2) `bidirectional_iterator_tag` if `I` satisfies `cpp17-bidirectional-iterator`, or otherwise
   (3.2.3.3) `forward_iterator_tag` if `I` satisfies `cpp17-forward-iterator`, or otherwise
   (3.2.3.4) `input_iterator_tag`.

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`.

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

```cpp
namespace std {
  template<class T>
```
requires is_object_v<T>
struct iterator_traits<T*>
{
    using iterator_concept = contiguous_iterator_tag;
    using iterator_category = random_access_iterator_tag;
    using value_type = remove_cv_t<T>;
    using difference_type = ptrdiff_t;
    using pointer = T*;
    using reference = T&;
};

6 [Example: 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

23.3.3.1 ranges::iter_move

The name ranges::iter_move denotes a customization point object (16.4.2.2.6). The expression ranges::iter_move(E) for some subexpression E is expression-equivalent to the following:

(1.1) — iter_move(E), if that expression is valid, with overload resolution performed in a context that does not include a declaration of ranges::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 with no diagnostic required.

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.

2 Let iter-exchange-move be the exposition-only function:

template<class X, class Y>
constexpr iter_value_t<remove_reference_t<X>> iter-exchange-move(X&& x, Y&& y)
    noexcept(noexcept(iter_value_t<remove_reference_t<X>>(iter_move(x))) &&
        noexcept(*x = iter_move(y)));

3 Effects: Equivalent to:

    iter_value_t<remove_reference_t<X>> old_value(iter_move(x));
    *x = iter_move(y);
    return old_value;

4 The expression ranges::iter_swap(E1, E2) for some subexpressions E1 and E2 is expression-equivalent to the following:
(4.1) — (void)iter_swap(E1, E2), if that expression is valid, 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 with no diagnostic required.

(4.2) — Otherwise, if the types of E1 and E2 each model `Readable`, and if the reference types of E1 and E2 model `SwappableWith` (18.4.9), then `ranges::swap(*E1, *E2)`.

(4.3) — Otherwise, if the types T1 and T2 of E1 and E2 model `IndirectlyMovableStorable<T1, T2>` and `IndirectlyMovableStorable<T2, T1>`, then (void)(*E1 = `iter-exchange-move`(E2, E1)), except that E1 is evaluated only once.

(4.4) — 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 [iterator.concepts]

23.3.4.1 General [iterator.concepts.general]

1 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>`.

(1.1) — If the qualified-id `ITER_TRAITS(I)::iterator_concept` is valid and names a type, then `ITER_CONCEPT(I)` denotes that type.

(1.2) — Otherwise, if the qualified-id `ITER_TRAITS(I)::iterator_category` is valid and names a type, then `ITER_CONCEPT(I)` denotes that type.

(1.3) — Otherwise, if `iterator_traits<I>` names a specialization generated from the primary template, then `ITER_CONCEPT(I)` denotes `random_access_iterator_tag`.

(1.4) — Otherwise, `ITER_CONCEPT(I)` does not denote a type.

2 [Note: `ITER_TRAITS` enables independent syntactic determination of an iterator’s category and concept. —end note] [Example]:

```cpp
struct I {
    using value_type = int;
    using difference_type = int;

    int operator*() const;
    I& operator++();
    I operator++(int);
    I& operator--();
    I operator--(int);

    bool operator==(I) const;
    bool operator!=(I) const;
};
```

`iter_traits<I>::iterator_category` denotes `input_iterator_tag`, and `ITER_CONCEPT(I)` denotes `random_access_iterator_tag`. —end example]

23.3.4.2 Concept Readable [iterator.concept-readable]

1 Types that are readable by applying `operator*` model the `Readable` concept, including pointers, smart pointers, and iterators.

```cpp
template<class In>
concept Readable =
    requires {
        typename iter_value_t<In>;
        typename iter_reference_t<In>;
        typename iter_rvalue_reference_t<In>;
    } &
```

§ 23.3.4.2
Given a value \( i \) of type \( I \), \( I \) models \texttt{Readable} only if the expression \( *i \) is equality-preserving. \( \textbf{[Note: The expression \( *i \) is indirectly required to be valid via the exposition-only \texttt{dereferenceable} concept (23.2). \textendnote]} \)

\section*{23.3.4.3 Concept \texttt{Writable}}

The \texttt{Writable} concept specifies the requirements for writing a value into an iterator's referenced object.

\begin{verbatim}
  template<class Out, class T>
  concept Writable =
    requires(Out&& o, T&& t) {
      *o = std::forward<T>(t);  // not required to be equality-preserving
      *std::forward<Out>(o) = std::forward<T>(t);  // not required to be equality-preserving
      const_cast<const iter_reference_t<Out>&&>(*o) =
        std::forward<T>(t);  // not required to be equality-preserving
      const_cast<const iter_reference_t<Out>&&>(*std::forward<Out>(o)) =
        std::forward<T>(t);  // not required to be equality-preserving
    };
\end{verbatim}

2 Let \( E \) be an an expression such that \texttt{decltype}((\( E \))) is \( T \), and let \( o \) be a dereferenceable object of type \( \text{Out} \). \( \text{Out} \) and \( T \) model \texttt{Writable<Out, T>} only if

\begin{enumerate}
  \item If \( \text{Out} \) and \( T \) model \texttt{Readable<Out> \&\& Same<iter_value_t<Out>, \texttt{decay_t<T>}>}, then \( *o \) after any above assignment is equal to the value of \( E \) before the assignment.
  \item After evaluating any above assignment expression, \( o \) is not required to be dereferenceable.
  \item If \( E \) is an xvalue (7.2.1), the resulting state of the object it denotes is valid but unspecified (16.5.5.15).
  \item \( \textbf{[Note: The only valid use of an \texttt{operator*} is on the left side of the assignment statement. Assignment through the same value of the writable type happens only once. \textendnote]} \)
  \item \texttt{Writable} has the awkward \texttt{const_cast} expressions to reject iterators with prvalue non-proxy reference types that permit rvalue assignment but do not also permit \texttt{const} rvalue assignment. Consequently, an iterator type \( I \) that returns \texttt{std::string} by value does not model \texttt{Writable<\( I \), std::string>}. \( \textbf{[end note]} \)
\end{enumerate}

\section*{23.3.4.4 Concept \texttt{WeaklyIncrementable}}

The \texttt{WeaklyIncrementable} 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 \texttt{EqualityComparable}.

\begin{verbatim}
  template<class I>
  concept WeaklyIncrementable =
    Semiregular<I> &&
    requires(I i) {
      typename iter_difference_t<I>;
      requires(SignedIntegral<iter_difference_t<I>>);
      { ++i } -> Same<I&>;  // not required to be equality-preserving
      i++;  // not required to be equality-preserving
    };
\end{verbatim}

2 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 \texttt{incrementable}. \( I \) models \texttt{WeaklyIncrementable<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 \texttt{addressof(++i)} is equal to \texttt{addressof(i)}.
  \item \( \textbf{[Note: For \texttt{WeaklyIncrementable} 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 never attempt to pass through the same incrementable value twice. They should be single-pass algorithms. These algorithms can be used with streams as the source of the input data through the \texttt{istream_iterator} class template. \textendnote]} \)
\end{enumerate}
23.3.4.5 Concept Incrementable

The **Incrementable** concept specifies requirements on types that can be incremented with the pre- and post-increment operators. The increment operations are required to be equality-preserving, and the type is required to be **EqualityComparable**. [Note: This supersedes the annotations on the increment expressions in the definition of **WeaklyIncrementable**. — end note]

```cpp
template<class I>
concept Incrementable =
    Regular<I> &&
    WeaklyIncrementable<I> &&
    requires(I i) {
        { i++ } -> Same<I>;
    };
```

Let *a* and *b* be incrementable objects of type *I*. *I* models **Incrementable** only if

(2.1) — If `bool(a == b)` then `bool(a++ == b)`.

(2.2) — If `bool(a == b)` then `bool(($(void)a++, a) == ++b)`.

[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 **Incrementable**. — end note]

23.3.4.6 Concept Iterator

The **Iterator** concept forms the basis of the iterator concept taxonomy; every iterator models **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).

```cpp
template<class I>
concept Iterator =
    requires(I i) {
        *i -> can-reference;
    } &&
    WeaklyIncrementable<I>;
```

23.3.4.7 Concept Sentinel

The **Sentinel** concept specifies the relationship between an **Iterator** type and a **Semiregular** type whose values denote a range.

```cpp
template<class S, class I>
concept Sentinel =
    Semiregular<S> &&
    Iterator<I> &&
    weakly-equality-comparable-with<S, I>; // See 18.5.3
```

Let *s* and *i* be values of type *S* and *I* such that [*i, s*) denotes a range. Types *S* and *I* model **Sentinel<S, I>** only if

(2.1) — *i == s* is well-defined.

(2.2) — If `bool(i != s)` then *i* is dereferenceable and [*++i, s*) denotes a range.

The domain of `==` is not static. Given an iterator *i* and sentinel *s* such that [*i, s*) denotes a range and *i* != *s*, *i* and *s* are not required to continue to denote a range after incrementing any other iterator equal to *i*. Consequently, *i == s* is no longer required to be well-defined.

23.3.4.8 Concept SizedSentinel

The **SizedSentinel** concept specifies requirements on an **Iterator** and a **Sentinel** that allow the use of the `−` operator to compute the distance between them in constant time.

```cpp
template<class S, class I>
concept SizedSentinel =
    Sentinel<S, I> &&
    !disable_sized_sentinel<remove_cv_t<S>, remove_cv_t<I>> &&
    requires(const I& i, const S& s) {
```
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{SizedSentinel}\}<\( I >\) only if

\begin{enumerate}
\item If \( N \) is representable by \text{iter_difference}\(_{<}\)\( I >\), then \( s - i \) is well-defined and equals \( N \).
\item If \(-N\) is representable by \text{iter_difference}\(_{<}\)\( I >\), then \( i - s \) is well-defined and equals \(-N\).
\end{enumerate}

[Note: \text{disable_sized_sentinel} allows use of sentinels and iterators with the library that satisfy but do not in fact model \text{SizedSentinel}. —end note]

[Example: The \text{SizedSentinel} concept is modeled by pairs of \text{RandomAccessIterator} (23.3.4.13) and by counted iterators and their sentinels (23.5.6.1). —end example]

### 23.3.4.9 Concept InputIterator

The InputIterator concept defines requirements for a type whose referenced values can be read (from the requirement for \text{Readable} (23.3.4.2)) and which can be both pre- and post-incremented. [Note: Unlike the \text{Cpp17InputIterator} requirements (23.3.5.2), the InputIterator concept does not need equality comparison since iterators are typically compared to sentinels. —end note]

\[
\text{template<class }I\text{>}
\text{concept InputIterator = }
\text{Iterator}<I> \&\&
\text{Readable}<I> \&\&
\text{requires } \{ \text{typename ITER_CONCEPT}(I); \} \&\&
\text{DerivedFrom<ITER_CONCEPT(I), input_iterator_tag>};
\]

### 23.3.4.10 Concept OutputIterator

The OutputIterator concept defines requirements for a type that can be used to write values (from the requirement for \text{Writable} (23.3.4.3)) and which can be both pre- and post-incremented. [Note: Output iterators are not required to model \text{EqualityComparable}. —end note]

\[
\text{template<class }I, \text{ class }T\text{>}
\text{concept OutputIterator = }
\text{Iterator}<I> \&\&
\text{Writable}<I, T> \&\&
\text{requires}(I i, T\& t) \{ \text{*i++ = std::forward<T>(t);} \text{; // not required to be equality-preserving} \};
\]

Let \( E \) be an expression such that \text{decltype}((E)) is \( T \), and let \( i \) be a dereferenceable object of type \( I \). \( I \) and \( T \) model OutputIterator\(<I, T\rangle \) only if \text{*i++ = E;} has effects equivalent to:

\[
\text{*i = E;}
\]

\[
\text{++i;}
\]

[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 ForwardIterator

The ForwardIterator concept adds equality comparison and the multi-pass guarantee, specified below.

\[
\text{template<class }I\text{>}
\text{concept ForwardIterator = }
\text{InputIterator}\(<I\)> \&\&
\text{DerivedFrom<ITER_CONCEPT(I), forward_iterator_tag> \&\&}
\text{Incrementable}<I> \&\&
\text{Sentinel}<I, I>;
\]

The domain of \( == \) for forward iterators is that of iterators over the same underlying sequence. However, value-initialized iterators of the same type may be compared and shall compare equal to other value-initialized iterators of the same type. [Note: Value-initialized iterators behave as if they refer past the end of the same empty sequence. —end note]
Pointers and references obtained from a forward iterator into a range \([i, s)\) shall remain valid while \([i, s)\) continues to denote a range.

Two dereferenceable iterators \(a\) and \(b\) of type \(X\) offer the multi-pass guarantee if:

1. \(a == b\) implies \(+a == +b\) and
2. The expression \((\text{void}[] X x\{++x;\}(a), *a)\) 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 BidirectionalIterator

The **BidirectionalIterator** concept adds the ability to move an iterator backward as well as forward.

```cpp
template<class I>
concept BidirectionalIterator = ForwardIterator<I> &&
   DerivedFrom<ITER_CONCEPT(I), bidirectional_iterator_tag> &&
   requires(I i) {
      { --i } -> Same<I&>;
      { i-- } -> Same<I>;
   };
```

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 \(--r--\).

Let \(a\) and \(b\) be equal objects of type \(I\). \(I\) models **BidirectionalIterator** only if:

1. If \(a\) and \(b\) are decrementable, then all of the following are true:
   1. \(\text{addressof}(-a) == \text{addressof}(a)\)
   2. \(\text{bool}(a-- == b)\)
   3. after evaluating both \(a--\) and \(--b\), \(\text{bool}(a == b)\) is still true
   4. \(\text{bool}(+-(-a) == b)\)
2. If \(a\) and \(b\) are incrementable, then \(\text{bool}(-(+a) == b)\).

### 23.3.4.13 Concept RandomAccessIterator

The **RandomAccessIterator** concept adds support for constant-time advancement with \(+=, +, -=,\) and \(-,\) as well as the computation of distance in constant time with \(-.\) Random access iterators also support array notation via subscripting.

```cpp
template<class I>
concept RandomAccessIterator = BidirectionalIterator<I> &&
   DerivedFrom<ITER_CONCEPT(I), random_access_iterator_tag> &&
   StrictTotallyOrdered<I> &&
   SizedSentinel<I, I> &&
   requires(I i, const I j, const iter_difference_t<I> n) {
      { i += n } -> Same<I&>;
      { j + n } -> Same<I>;
      { n + j } -> Same<I>;
      { i -= n } -> Same<I&>;
      { j - n } -> Same<I>;
      { j[n] } -> Same<iter_reference_t<I>>;
   };
```

Let \(a\) and \(b\) be valid iterators of type \(I\) such that \(b\) is reachable from \(a\) after \(n\) applications of \;++a,\) let \(D\) be \(\text{iter_difference_t}<I>\), and let \(n\) denote a value of type \(D\). \(I\) models **RandomAccessIterator** only if

1. \((a += n)\) is equal to \(b\).
2. \(\text{addressof}(a += n)\) is equal to \(\text{addressof}(a)\).
3. \((a + n)\) is equal to \((a += n)\).
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 \(((a + x) + y)\).

§ 23.3.4.13
(2.5) \((a + D(0))\) is equal to \(a\).

(2.6) If \((a + D(n - 1))\) is valid, then \((a + n)\) is equal to \(+(a + D(n - 1))\).

(2.7) \((b += -n)\) 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 \(addressof(b)\).

(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 ContiguousIterator

The `ContiguousIterator` concept provides a guarantee that the denoted elements are stored contiguously in memory.

```cpp
template<class I>
concept ContiguousIterator =
    RandomAccessIterator<I> &&
    DerivedFrom<ITER_CONCEPT(I), contiguous_iterator_tag> &&
    is_lvalue_reference_v<iter_reference_t<I>> &&
    Same<iter_value_t<I>, remove_cvref_t<iter_reference_t<I>>>;
```

(2) Let \(a\) and \(b\) be dereferenceable iterators of type \(I\) such that \(b\) is reachable from \(a\), and let \(D\) be `iter_difference_t<I>`. The type \(I\) models `ContiguousIterator` only if \(addressof(*(a + D(b - a)))\) is equal to \(addressof(*a) + D(b - a)\).

### 23.3.5 C++17 iterator requirements

The `Cpp17Iterator` requirements form the basis of the iterator taxonomy; every iterator satisfies 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).

A type \(X\) satisfies the `Cpp17Iterator` requirements if:

(2.1) \(X\) satisfies the `Cpp17CopyConstructible`, `Cpp17CopyAssignable`, and `Cpp17Destructible` requirements (16.5.3.1) and `iter_value_t<X>` is swappable (16.5.3.2), and

(2.2) the expressions in Table 73 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>(\ast r)</td>
<td>unspecified</td>
<td>(X&amp;)</td>
<td>(Expects: r) is dereferenceable.</td>
</tr>
<tr>
<td>(++r)</td>
<td></td>
<td>(\ast r)</td>
<td></td>
</tr>
</tbody>
</table>

### 23.3.5.2 Input iterators

A class or pointer type \(X\) satisfies the requirements of an input iterator for the value type \(T\) if \(X\) satisfies the `Cpp17Iterator` (23.3.5.1) and `Cpp17EqualityComparable` (Table 23) requirements and the expressions in Table 74 are valid and have the indicated semantics.

In Table 74, 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 74 — Cpp17InputIterator requirements (in addition to Cpp17Iterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>a != b</td>
<td>contextually convertible to bool</td>
<td>!(a == b)</td>
<td>Expects: (a, b) is in the domain of ==.</td>
</tr>
<tr>
<td>*a</td>
<td>reference, convertible to T</td>
<td>*a is dereferenceable. The expression (void)*a, *a is equivalent to *a. If a == b and (a, b) is in the domain of == then *a is equivalent to *b.</td>
<td></td>
</tr>
<tr>
<td>a-&gt;m</td>
<td>(*a).m</td>
<td>a is dereferenceable.</td>
<td></td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td>r is dereferenceable. Ensures: r is incrementable. Remarks: After this operation r is not required to be dereferenceable.</td>
<td></td>
</tr>
<tr>
<td>(void)r++</td>
<td>convertible to T</td>
<td>equivalent to (void)++r</td>
<td></td>
</tr>
<tr>
<td>*r++</td>
<td>convertible to T</td>
<td>{ T tmp = *r; ++r; return tmp; }</td>
<td></td>
</tr>
</tbody>
</table>

3 [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 29). These algorithms can be used with istreams 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 satisfies the requirements of an output iterator if X satisfies the Cpp17Iterator requirements (23.3.5.1) and the expressions in Table 75 are valid and have the indicated semantics.

Table 75 — 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>*r = o</td>
<td>result is not used</td>
<td>Remarks: After this operation r is not required to be dereferenceable. Ensures: r is incrementable.</td>
<td></td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td>addressof(r) == addressof(++r). Remarks: After this operation r is not required to be dereferenceable. Ensures: r is incrementable.</td>
<td></td>
</tr>
</tbody>
</table>

§ 23.3.5.3
Table 75 — `Cpp17OutputIterator` requirements (in addition to `Cpp17Iterator`) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>r++</td>
<td>convertible to <code>const X&amp;</code></td>
<td><code>{ X tmp = r; ++r; return tmp; }</code></td>
<td>Remarks: After this operation <code>r</code> is not required to be dereferenceable. Ensures: <code>r</code> is incrementable.</td>
</tr>
<tr>
<td>= o</td>
<td>result is not used</td>
<td>Remarks: After this operation <code>r</code> is not required to be dereferenceable. Ensures: <code>r</code> is incrementable.</td>
<td></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` satisfies the requirements of a forward iterator if

1. `X` satisfies the `Cpp17InputIterator` requirements (23.3.5.2),
2. `X` satisfies 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 76 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]

Two dereferenceable iterators `a` and `b` of type `X` offer the multi-pass guarantee if:

1. `a == b` implies `++a == ++b` and
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 76 — `Cpp17ForwardIterator` requirements (in addition to `Cpp17InputIterator`)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>r++</td>
<td>convertible to <code>const X&amp;</code></td>
<td><code>{ X tmp = r; ++r; return tmp; }</code></td>
<td></td>
</tr>
<tr>
<td>= o</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

1 A class or pointer type X satisfies the requirements of a bidirectional iterator if, in addition to satisfying the Cpp17ForwardIterator requirements, the following expressions are valid as shown in Table 77.

Table 77 — Cpp17BidirectionalIterator requirements (in addition to Cpp17ForwardIterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>--r</td>
<td>X&amp;</td>
<td>Expects: there exists s such that r == ++s. Ensures: r is dereferenceable. --(++r) == r. --r == --s implies r == s. addressof(r) == addressof(--r).</td>
<td></td>
</tr>
<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>

2 [Note: Bidirectional iterators allow algorithms to move iterators backward as well as forward. — end note]

23.3.5.6 Random access iterators

1 A class or pointer type X satisfies the requirements of a random access iterator if, in addition to satisfying the Cpp17BidirectionalIterator requirements, the following expressions are valid as shown in Table 78.

Table 78 — Cpp17RandomAccessIterator requirements (in addition to Cpp17BidirectionalIterator)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>
</tr>
<tr>
<td>a + n</td>
<td>X</td>
<td>{ X tmp = a; return tmp += n; }</td>
<td>a + n == n + a.</td>
</tr>
<tr>
<td>n + a</td>
<td>X</td>
<td>{ X tmp = a; return tmp += n; }</td>
<td></td>
</tr>
<tr>
<td>r -= n</td>
<td>X&amp;</td>
<td>return r += -n;</td>
<td>Expects: the absolute value of n is in the range of representable values of difference_type.</td>
</tr>
<tr>
<td>a - n</td>
<td>X</td>
<td>{ X tmp = a; return tmp -= n; }</td>
<td></td>
</tr>
<tr>
<td>b - a</td>
<td>difference_-type</td>
<td>return n</td>
<td>Expects: there exists a value n of type difference_type such that a + n == b. b == a + (b - a).</td>
</tr>
<tr>
<td>a[n]</td>
<td>convertible to</td>
<td>*(a + n)</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>
</tr>
</tbody>
</table>
Table 78 — Cpp17RandomAccessIterator requirements (in addition to Cpp17BidirectionalIterator) (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>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>
</tr>
<tr>
<td>a &gt;= b</td>
<td>contextually convertible to bool</td>
<td>!(a &lt; b)</td>
<td></td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>contextually convertible to bool</td>
<td>!(a &gt; b)</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 {
    template<class F, class I>
    concept IndirectUnaryInvocable = 
        Readable<I> &&
        CopyConstructible<F> &&
        Invocable<F&, iter_value_t<I>&> &&
        Invocable<F&, iter_reference_t<I>> &&
        Invocable<F&, iter_common_reference_t<I>> &&
        CommonReference<
            invoke_result_t<F&, iter_value_t<I>&>,
            invoke_result_t<F&, iter_reference_t<I>>;>

    template<class F, class I>
    concept IndirectRegularUnaryInvocable = 
        Readable<I> &&
        CopyConstructible<F> &&
        RegularInvocable<F&, iter_value_t<I>&> &&
        RegularInvocable<F&, iter_reference_t<I>> &&
        RegularInvocable<F&, iter_common_reference_t<I>> &&
        CommonReference<
            invoke_result_t<F&, iter_value_t<I>&>,
            invoke_result_t<F&, iter_reference_t<I>>;>

    template<class F, class I>
    concept IndirectUnaryPredicate = 
        Readable<I> &&
        CopyConstructible<F> &&
        Predicate<F&, iter_value_t<I>&> &&
        Predicate<F&, iter_reference_t<I>> &&
        Predicate<F&, iter_common_reference_t<I>>;

    template<class F, class I1, class I2 = I1>
    concept IndirectRelation = 
        Readable<I1> && Readable<I2> &&
        CopyConstructible<F> &&
        Relation<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
        Relation<F&, iter_value_t<I1>&, iter_reference_t<I2>> &&
```
Relation<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
Relation<F&, iter_reference_t<I1>, iter_reference_t<I2>>&
Relation<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

template<class F, class I1, class I2 = I1>
concept IndirectStrictWeakOrder =
Readable<I1> && Readable<I2> &&
CopyConstructible<F> &&
StrictWeakOrder<F&, iter_value_t<I1>&, iter_value_t<I2>&> &&
StrictWeakOrder<F&, iter_value_t<I1>&, iter_reference_t<I2>> &&
StrictWeakOrder<F&, iter_reference_t<I1>, iter_value_t<I2>&> &&
StrictWeakOrder<F&, iter_reference_t<I1>, iter_reference_t<I2>> &&
StrictWeakOrder<F&, iter_common_reference_t<I1>, iter_common_reference_t<I2>>;

23.3.6.3 Class template projected

Class template projected is used to constrain algorithms that accept callable objects and projections (16.3.21).

It combines a Readable type I and a callable object type Proj into a new Readable type whose reference type is the result of applying Proj to the iter_reference_t of I.

namespace std {
    template<Readable I, IndirectRegularUnaryInvocable<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<WeaklyIncrementable 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

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 Readable and Writable types: IndirectlyMovable, IndirectlyCopyable, and IndirectlySwappable. There are three relational concepts for rearrangements: Permutable, Mergeable, and Sortable. There is one relational concept for comparing values from different sequences: IndirectlyComparable.

[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 IndirectlyMovable

The IndirectlyMovable concept specifies the relationship between a Readable type and a Writable type between which values may be moved.

template<class In, class Out>
concept IndirectlyMovable =
Readable<In> &&
Writable<Out, iter_rvalue_reference_t<In>>;

The IndirectlyMovableStorable concept augments IndirectlyMovable with additional requirements enabling the transfer to be performed through an intermediate object of the Readable type’s value type.

template<class In, class Out>
concept IndirectlyMovableStorable =
IndirectlyMovable<In, Out> &&
Writable<Out, iter_value_t<In>> &&
Movable<iter_value_t<In>> &&
Constructible<iter_value_t<In>, iter_rvalue_reference_t<In>> &&
Assignable<iter_value_t<In>&, iter_rvalue_reference_t<In>>;
3 Let \( i \) be a dereferenceable value of type \( \text{In} \). \( \text{In} \) and \( \text{Out} \) model \text{IndirectlyMovableStorable}\(<\text{In}, \ \text{Out}>\) only if after the initialization of the object \( \text{obj} \) in
\[
\text{iter\_value\_t<}\text{In}\text{> obj(ranges::iter\_move(}i\text{));}
\]
\( \text{obj} \) is equal to the value previously denoted by \( \ast i \). If \( \text{iter\_rvalue\_reference\_t<}\text{In}>\) is an rvalue reference type, the resulting state of the value denoted by \( \ast i \) is valid but unspecified (16.5.5.15).

23.3.7.3 Concept IndirectlyCopyable 

The \text{IndirectlyCopyable} concept specifies the relationship between a \text{Readable} type and a \text{Writable} type between which values may be copied.

\[
\text{template<class } \text{In}, \text{ class } \text{Out}\text{>}
\text{concept } \text{IndirectlyCopyable } =
\text{Readable<}\text{In}\text{> } \&\&
\text{Writable<}\text{Out}, \text{ const } \text{iter\_value\_t<}\text{In}>\text{&}\&
\text{Copyable<}\text{iter\_value\_t<}\text{In}>\text{&}\&
\text{Constructible<}\text{iter\_value\_t<}\text{In}>\text{, iter\_reference\_t<}\text{In}>>\text{&}\&
\text{Assignable<}\text{iter\_value\_t<}\text{In}>\&\text{, iter\_reference\_t<}\text{In}>>;
\]

23.3.7.4 Concept IndirectlySwappable 

The \text{IndirectlySwappable} concept specifies a swappable relationship between the values referenced by two \text{Readable} types.

\[
\text{template<class } \text{I1}, \text{ class } \text{I2 } = \text{I1}\text{>}
\text{concept } \text{IndirectlySwappable } =
\text{Readable<}\text{I1}\text{> } \&\& \text{Readable<}\text{I2}\text{> } \&\&
\text{requires(I1& i1, I2& i2) } \{
\text{ranges::iter\_swap(i1, i1);}
\text{ranges::iter\_swap(i2, i2);}
\text{ranges::iter\_swap(i1, i2);}
\text{ranges::iter\_swap(i2, i1);}
\};
\]

23.3.7.5 Concept IndirectlyComparable 

The \text{IndirectlyComparable} concept specifies the common requirements of algorithms that compare values from two different sequences.

\[
\text{template<class } \text{I1}, \text{ class } \text{I2}, \text{ class } \text{R}, \text{ class } \text{P1 } = \text{identity}, \text{ class } \text{P2 } = \text{identity}\text{>}
\text{concept } \text{IndirectlyComparable } =
\text{IndirectRelation<}\text{R}, \text{ projected<}\text{I1}, \text{P1}>\text{, projected<}\text{I2}, \text{P2}>>;
\]

23.3.7.6 Concept Permutable 

The \text{Permutable} concept specifies the common requirements of algorithms that reorder elements in place by moving or swapping them.
template<class I>
concept Permutable =
   ForwardIterator<I> &&
   IndirectlyMovableStorable<I, I> &&
   IndirectlySwappable<I, I>;

23.3.7.7 Concept Mergeable

The Mergeable concept specifies the requirements of algorithms that merge sorted sequences into an output sequence by copying elements.

template<class I1, class I2, class Out, class R = ranges::less,
   class P1 = identity, class P2 = identity>
concept Mergeable =
   InputIterator<I1> &&
   InputIterator<I2> &&
   WeaklyIncrementable<Out> &&
   IndirectlyCopyable<I1, Out> &&
   IndirectlyCopyable<I2, Out> &&
   IndirectlyStrictWeakOrder<R, projected<I1, P1>, projected<I2, P2>>;

23.3.7.8 Concept Sortable

The Sortable concept specifies the common requirements of algorithms that permute sequences into ordered sequences (e.g., sort).

template<class I, class R = ranges::less, class P = identity>
concept Sortable =
   Permutable<I> &&
   IndirectStrictWeakOrder<R, projected<I, P>>;

23.4 Iterator primitives

To simplify the use of iterators, the library provides several classes and functions.

23.4.1 Standard iterator tags

It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the library introduces category tag classes which are used as compile time tags for algorithm selection. They are: output_iterator_tag, input_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag, random_access_iterator_tag, and contiguous_iterator_tag. For every iterator of type I, iterator_traits<I>::iterator_category shall be defined to be a category tag that describes the iterator’s behavior. Additionally, iterator_traits<I>::iterator_concept may be used to indicate conformance to the iterator concepts (23.3.4).

namespace std {
   struct output_iterator_tag { };  
   struct input_iterator_tag { };    
   struct forward_iterator_tag: public input_iterator_tag { }; 
   struct bidirectional_iterator_tag: public forward_iterator_tag { }; 
   struct random_access_iterator_tag: public bidirectional_iterator_tag { }; 
   struct contiguous_iterator_tag: public random_access_iterator_tag { }; 
}

[Example: For a program-defined iterator BinaryTreeIterator, it could be included into the bidirectional iterator category by specializing the iterator_traits template:

    template<class T> struct iterator_traits<BinaryTreeIterator<T>> {
       using iterator_category = bidirectional_iterator_tag;
       using difference_type = ptrdiff_t;
       using value_type = T;
       using pointer = T*;
       using reference = T&;
    };

— end example]

[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:}
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);

Expects: 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);

Expects: 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 RandomAccessIterator forward n steps in constant time. For an iterator type that does not model RandomAccessIterator, 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 (6.4.2). When found by unqualified (6.4.1) name lookup for the postfix-expression in a function call (7.6.1.2), they inhibit argument-dependent name lookup.
[Example:
void foo() {
    using namespace std::ranges;
    std::vector<int> vec{1,2,3};
    distance(begin(vec), end(vec));  // #1
}
]
The function call expression at #1 invokes std::ranges::distance, not std::distance, despite that (a) the iterator type returned from begin(vec) and end(vec) may be associated with namespace std and (b) std::distance is more specialized (13.6.6.2) than std::ranges::distance since the former requires its first two parameters to have the same type. —end example]

3 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<Iterator I>
constexpr void ranges::advance(I& i, iter_difference_t<I> n);
```

1 Expects: If I does not model BidirectionalIterator, n is not negative.
2 Effects:
(2.1) — If I models RandomAccessIterator, equivalent to i += n.
(2.2) — Otherwise, if n is non-negative, increments i by n.
(2.3) — Otherwise, decrements i by -n.

```cpp
template<Iterator I, Sentinel<I> S>
constexpr void ranges::advance(I& i, S bound);
```

3 Expects: [i, bound) denotes a range.
4 Effects:
(4.1) — If I and S model Assignable<I&, S>, equivalent to i = std::move(bound).
(4.2) — Otherwise, if S and I model SizedSentinel<S, I>, equivalent to ranges::advance(i, bound - i).
(4.3) — Otherwise, while bool(i != bound) is true, increments i.

```cpp
template<Iterator I, Sentinel<I> S>
constexpr iter_difference_t<I> ranges::advance(I& i, iter_difference_t<I> n, S bound);
```

5 Expects: 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 BidirectionalIterator, and I and S model Same<I, S>.
6 Effects:
(6.1) — If S and I model SizedSentinel<S, I>:
(6.1.1) — If |n| ≥ |bound - i|, equivalent to ranges::advance(i, bound).
(6.1.2) — Otherwise, equivalent to ranges::advance(i, n).
(6.2) — Otherwise,
(6.2.1) — if n is non-negative, while bool(i != bound) is true, increments i but at most n times.
(6.2.2) — Otherwise, while bool(i != bound) is true, decrements i but at most -n times.
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<Iterator I, Sentinel<I> S>
constexpr iter_difference_t<I> ranges::distance(I first, S last);
```

1 Expects: [first, last) denotes a range, or [last, first) denotes a range and S and I model Same<S, I> && SizedSentinel<S, I>.

§ 23.4.3.2
Effects: If $S$ and $I$ model $\text{SizedSentinel}<S, I>$, returns $(\text{last} - \text{first})$; otherwise, returns the number of increments needed to get from $\text{first}$ to $\text{last}$.

```cpp
template<Range R>
constexpr iter_difference_t<iterator_t<R>> ranges::distance(R&& r);
```

Effects: If $R$ models $\text{SizedRange}$, equivalent to:
```
return 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

```cpp
template<Iterator I>
constexpr I ranges::next(I x);
```

Effects: Equivalent to:
```
++x; return x;
```

```cpp
template<Iterator I>
constexpr I ranges::next(I x, iter_difference_t<I> n);
```

Effects: Equivalent to:
```
ranges::advance(x, n); return x;
```
// 24.3

```cpp
template<Iterator I, Sentinel<I> S>
constexpr I ranges::next(I x, S bound);
```

Effects: Equivalent to:
```
ranges::advance(x, bound); return x;
```
// 24.3

```cpp
template<Iterator I, Sentinel<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;
```
// 24.3

23.4.3.4 ranges::prev

```cpp
template<BidirectionalIterator I>
constexpr I ranges::prev(I x);
```

Effects: Equivalent to:
```
--x; return x;
```

```cpp
template<BidirectionalIterator I>
constexpr I ranges::prev(I x, iter_difference_t<I> n);
```

Effects: Equivalent to:
```
ranges::advance(x, -n); return x;
```
// 24.3

```cpp
template<BidirectionalIterator I>
constexpr I ranges::prev(I x, iter_difference_t<I> n, I bound);
```

Effects: Equivalent to:
```
ranges::advance(x, -n, bound); return x;
```
// 24.3

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`

```cpp
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>;
  }
} // namespace std
```

§ 23.5.1.1 906
constexpr reverse_iterator();
constexpr explicit reverse_iterator(Iterator x);
template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);

caseexpr Iterator base() const;
caseexpr reference operator*() const;
caseexpr pointer operator->() const requires see below;

caseexpr reverse_iterator& operator++();
caseexpr reverse_iterator operator++(int);
caseexpr reverse_iterator& operator--();
caseexpr reverse_iterator operator--(int);

caseexpr reverse_iterator operator+(difference_type n) const;
caseexpr reverse_iterator& operator+=(difference_type n);
caseexpr reverse_iterator operator-(difference_type n) const;
caseexpr reverse_iterator& operator-=(difference_type n);

caseexpr unspecified operator[](difference_type n) const;

friend constexpr iter_rvalue_reference_t<Iterator>
    iter_move(const reverse_iterator& i) noexcept(see below);

friend constexpr void
    iter_swap(const reverse_iterator& x,
              const reverse_iterator<Iterator2>& y) noexcept(see below);

protected:
    Iterator current;
};

The member typedef-name iterator_concept denotes
— random_access_iterator_tag if Iterator models RandomAccessIterator, and
— bidirectional_iterator_tag otherwise.

The member typedef-name iterator_category denotes
— random_access_iterator_tag if the type iterator_traits<Iterator>::iterator_category models
    DerivedFrom<random_access_iterator_tag>, and
— iterator_traits<Iterator>::iterator_category otherwise.

23.5.1.2 Requirements

The template parameter Iterator shall either meet the requirements of a Cpp17BidirectionalIterator (23.3.5.5)
    or model BidirectionalIterator (23.3.4.12).

Additionally, Iterator shall either meet the requirements of a Cpp17RandomAccessIterator (23.3.5.6) or
    model RandomAccessIterator (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 (13.8.1).

23.5.1.3 Construction and assignment

constexpr reverse_iterator();

Effects: Value-initializes current. Iterator operations applied to the resulting iterator have defined
    behavior if and only if the corresponding operations are defined on a value-initialized iterator of type
    Iterator.
constexpr explicit reverse_iterator(Iterator x);

Effects: Initializes current with x.

template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);

Effects: Initializes current with u.current.

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

constexpr Iterator base() const; // explicit

Returns: current.

23.5.1.5 Element access

constexpr reference operator*() const;

Effects: As if by:
    Iterator tmp = current;
    return *--tmp;

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->();

constexpr unspecified operator[](difference_type n) const;

Returns: current[-n-1].

23.5.1.6 Navigation

constexpr reverse_iterator operator+(difference_type n) const;

Returns: reverse_iterator(current+n).

constexpr reverse_iterator operator-(difference_type n) const;

Returns: reverse_iterator(current+n).

constexpr reverse_iterator& operator++();

Effects: As if by: --current;

Returns: *this.

constexpr reverse_iterator operator++(int);

Effects: As if by:
    reverse_iterator tmp = *this;
    --current;
    return tmp;

constexpr reverse_iterator& operator--();

Effects: As if by ++current.

Returns: *this.

§ 23.5.1.6 908
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

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().

§ 23.5.1.7
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 rvalue_reference_t<Iterator>
iter_move(const reverse_iterator& i) noexcept;

Effects: Equivalent to:
    auto tmp = i.base();
    return ranges::iter_move(--tmp);

Remarks: The expression in noexcept is equivalent to:
    is_nothrow_copy_constructible_v<Iterator> &&
    noexcept(ranges::iter_move(--declval<Iterator&>()))
```

```
template<IndirectlySwappable<Iterator> Iterator2>
friend constexpr void
iter_swap(const reverse_iterator& x,
          const reverse_iterator<Iterator2>& y) noexcept;

Effects: Equivalent to:
    auto xtmp = x.base();
    auto ytmp = y.base();
    ranges::iter_swap(--xtmp, --ytmp);

Remarks: The expression in noexcept is equivalent to:
    is_nothrow_copy_constructible_v<Iterator> &&
    is_nothrow_copy_constructible_v<Iterator2> &&
    noexcept(ranges::iter_swap(--declval<Iterator>&)(), --declval<Iterator2&>())
```

```
template<class Iterator>
constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);

Returns: reverse_iterator<Iterator>(i).
```

23.5.2 Insert iterators

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 satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.
23.5.2.1 Class template 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);
1 Effects: Initializes container with addressof(x).

constexpr back_insert_iterator& operator=(const typename Container::value_type& value);
2 Effects: As if by: container->push_back(value);
3 Returns: *this.

constexpr back_insert_iterator& operator=(typename Container::value_type&& value);
4 Effects: As if by: container->push_back(std::move(value));
5 Returns: *this.

constexpr back_insert_iterator& operator*();
6 Returns: *this.

constexpr back_insert_iterator& operator++();
7 Returns: *this.

23.5.2.2 back_inserter

template<class Container>
constexpr back_insert_iterator<Container> back_inserter(Container& x);
1 Returns: back_insert_iterator<Container>(x).

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(Container& x) 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);

Effects: Initializes container with addressof(x).

constexpr front_insert_iterator& operator=(const typename Container::value_type& value);

Effects: As if by: container->push_front(value);

Returns: *this.

constexpr front_insert_iterator& operator=(typename Container::value_type&& value);

Effects: As if by: container->push_front(std::move(value));

Returns: *this.

constexpr front_insert_iterator& operator*();

Returns: *this.

constexpr front_insert_iterator& operator++();
constexpr front_insert_iterator& operator++(int);

Returns: *this.

23.5.2.2.2 front_inserter

template<class Container>
constexpr front_insert_iterator<Container> front_inserter(Container& x);

Returns: front_insert_iterator<Container>(x).

23.5.2.3 Class template insert_iterator

namespace std {
    template<class Container>
    class insert_iterator {
    protected:
        Container* container = nullptr;
        iterator_t<Container> iter = iterator_t<Container>();
    public:
        using iterator_category = output_iterator_tag;
        using value_type = void;
        using difference_type = ptrdiff_t;
        using pointer = void;
        using reference = void;
        using container_type = Container;
    };

§ 23.5.2.3 912
```cpp
insert_iterator() = default;
constexpr insert_iterator(Container& x, 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

```cpp
constexpr insert_iterator(Container& x, iterator_t<Container> i);
```

**Effects:** Initializes container with `addressof(x)` and iter with i.

```cpp
constexpr insert_iterator& operator=(const typename Container::value_type& value);
```

**Effects:** As if by:
```cpp
iter = container->insert(iter, value);
++iter;
```

**Returns:** *this.

```cpp
constexpr insert_iterator& operator=(typename Container::value_type&& value);
```

**Effects:** As if by:
```cpp
iter = container->insert(iter, std::move(value));
++iter;
```

**Returns:** *this.

```cpp
constexpr insert_iterator& operator*();
```

**Returns:** *this.

```cpp
constexpr insert_iterator& operator++();
constexpr insert_iterator& operator++(int);
```

**Returns:** *this.

### 23.5.2.3.2 inserter

```cpp
template<class Container>
constexpr insert_iterator<Container> inserter(Container& x, iterator_t<Container> i);
```

**Returns:** insert_iterator<Container>(x, i).

### 23.5.3 Move iterators and sentinels

**Example:**
```cpp
list<string> s;
// populate the list s
vector<string> v1(s.begin(), s.end()); // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
    make_move_iterator(s.end())); // moves strings into v2
```

— end example]
23.5.3.1 Class template move_iterator

namespace std {
    template<class Iterator>
    class move_iterator {
    public:
        using iterator_type = Iterator;
        using iterator_concept = input_iterator_tag;
        using iterator_category = see below;
        using value_type = iter_value_t<Iterator>;
        using difference_type = iter_difference_t<Iterator>;
        using pointer = Iterator;
        using reference = iter_rvalue_reference_t<Iterator>;

        constexpr move_iterator();
        constexpr explicit move_iterator(Iterator i);
        template<class U> constexpr move_iterator(const move_iterator<U>& u);
        template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);
        constexpr iterator_type base() const;
        constexpr 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<Iterator> S>
        friend constexpr bool operator==(const move_iterator& x, const move_sentinel<S>& y);
        template<Sentinel<Iterator> S>
        friend constexpr bool operator==(const move_sentinel<S>& x, const move_iterator& y);
        template<Sentinel<Iterator> S>
        friend constexpr bool operator!=(const move_iterator& x, const move_sentinel<S>& y);
        template<Sentinel<Iterator> S>
        friend constexpr bool operator!=(const move_sentinel<S>& x, const move_iterator& y);
        template<SizedSentinel<Iterator> S>
        friend constexpr iter_difference_t<Iterator> operator-(const move_sentinel<S>& x, const move_iterator& y);
        template<SizedSentinel<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<IndirectlySwappable<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 DerivedFrom<random_access_iterator_tag>, and

(1.2) — iterator_traits<Iterator>::iterator_category otherwise.

23.5.3.2 Requirements

The template parameter Iterator shall either meet the Cpp17InputIterator requirements (23.3.5.2) or model InputIterator (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 BidirectionalIterator (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 RandomAccessIterator (23.3.4.13).

23.5.3.3 Construction and assignment

constexpr move_iterator();
1 Effects: Constructs a move_iterator, value-initializing current. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a value-initialized iterator of type Iterator.

conestexpr explicit move_iterator(Iterator i);
2 Effects: Constructs a move_iterator, initializing current with i.

template<class U> constexpr move_iterator(const move_iterator<U>& u);
3 Mandates: U is convertible to Iterator.
4 Effects: Constructs a move_iterator, initializing current with u.base().

template<class U> constexpr move_iterator& operator=(const move_iterator<U>& u);
5 Effects: Assigns u.base() to current.

23.5.3.4 Conversion

constexpr Iterator base() const;
1 Returns: current.

23.5.3.5 Element access

constexpr reference operator*() const;
1 Effects: Equivalent to: return ranges::iter_move(current);

constexpr reference operator[](difference_type n) const;
2 Effects: Equivalent to: ranges::iter_move(current + n);

23.5.3.6 Navigation

constexpr move_iterator& operator++();
1 Effects: As if by ++current.
2 Returns: *this.

constexpr auto operator++(int);
3 Effects: If Iterator models ForwardIterator, 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<Iterator> S>
friend constexpr bool operator==(const move_iterator& x,
 const move_sentinel<S>& y);

template<Sentinel<Iterator> S>
friend constexpr bool operator==(const move_sentinel<S>& x,
 const move_iterator& 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);

template<Sentinel<Iterator> S>
friend constexpr bool operator!=(const move_iterator& x,
 const move_sentinel<S>& y);

template<Sentinel<Iterator> S>
friend constexpr bool operator!=(const move_sentinel<S>& x,
 const move_iterator& y);
 Constraints: x.base() == y.base() is well-formed and convertible to bool.
 Returns: !(x == y).

template<class Iterator1, class Iterator2>
constexpr bool operator<(const move_iterator<Iterator1>& x,
 const move_iterator<Iterator2>& y);

template<Sentinel<Iterator> S>
friend constexpr bool operator<(const move_iterator& x,
 const move_sentinel<S>& y);

template<Sentinel<Iterator> S>
friend constexpr bool operator<(const move_sentinel<S>& x,
 const move_iterator& 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);

template<Sentinel<Iterator> S>
friend constexpr bool operator>(const move_iterator& x,
 const move_sentinel<S>& y);

template<Sentinel<Iterator> S>
friend constexpr bool operator>(const move_sentinel<S>& x,
 const move_iterator& y);
 Constraints: y.base() < x.base() is well-formed and convertible to bool.
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).

23.5.3.8 Non-member functions

template<class Iterator1, class Iterator2>
constexpr auto operator-(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y)
-> decltype(x.base() - y.base());

template<SizedSentinel<Iterator> S>
friend constexpr iter_difference_t<Iterator>
operator-(const move_sentinel<S>& x, const move_iterator<y>);

template<SizedSentinel<Iterator> S>
friend constexpr iter_difference_t<Iterator>
operator-(const move_iterator<y>, const move_sentinel<S>& y);

Returns: x.base() - y.base().

template<class Iterator>
constexpr move_iterator<Iterator> operator+(iter_difference_t<Iterator> n, const move_iterator<Iterator>& x);

Constraints: x + n is well-formed and has type Iterator.

Returns: x + n.

friend constexpr iter_rvalue_reference_t<Iterator>
iter_move(const move_iterator& i)
noexcept(noexcept(ranges::iter_move(i.current)));

Effects: Equivalent to: return ranges::iter_move(i.current);

template<IndirectlySwappable<Iterator> Iterator2>
friend constexpr void
iter_swap(const move_iterator<y>, const move_iterator<Iterator2>& y)
noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

Effects: Equivalent to: ranges::iter_swap(x.current, y.current).

template<class Iterator>
constexpr move_iterator<Iterator> make_move_iterator(Iterator i);

Returns: move_iterator<Iterator>(i).

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<S, I>, move_sentinel<S> and move_iterator<I> model Sentinel<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:]

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, 
IndirectUnaryPredicate<I> Pred>
requires IndirectlyMoveable<I, O>
void move_if(I first, S last, O out, Pred pred) {
    std::ranges::copy_if(move_iterator<I>{first}, move_sentinel<S>{last}, out, pred);
}
namespace std {
    template<Semiregular S>
    class move_sentinel {
        public:
            constexpr move_sentinel();
            constexpr explicit move_sentinel(S s);
            template<class S2>
                requires ConvertibleTo<const S2&, S>
                constexpr move_sentinel(const move_sentinel<S2>& s);
            template<class S2>
                requires Assignable<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 [move.sent.ops]

constexpr move_sentinel();
1 Effects: Value-initializes last. If is_trivially_default_constructible_v<S> is true, then this constructor is a constexpr constructor.

constexpr explicit move_sentinel(S s);
2 Effects: Initializes last with std::move(s).

template<class S2>
    requires ConvertibleTo<const S2&, S>
    constexpr move_sentinel(const move_sentinel<S2>& s);
3 Effects: Initializes last with s.last.

template<class S2>
    requires Assignable<S&, const S2&>
    constexpr move_sentinel& operator=(const move_sentinel<S2>& s);
4 Effects: Equivalent to: last = s.last; return *this;

23.5.4 Common iterators [iterators.common]

23.5.4.1 Class template common_iterator [common.iterator]

1 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 s
    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<Iterator I, Sentinel<I> S>
    requires (!Same<I, S>)
    class common_iterator {
    public:
        constexpr common_iterator() = default;
        constexpr common_iterator(I i);
        constexpr common_iterator(S s);
        template<class I2, class S2>
        requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S>
        constexpr common_iterator(const common_iterator<I2, S2>& x);
    
        template<class I2, class S2>
        requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S> &&
        Assignable<I&, const I2&> && Assignable<S&, const S2&>
        common_iterator& operator=(const common_iterator<I2, S2>& x);
    
        decltype(auto) operator*();
        decltype(auto) operator*() const
        requires dereferenceable<const I>;
        decltype(auto) operator->() const
        requires see below;
    
        common_iterator& operator++();
        decltype(auto) operator++(int);
    
        template<class I2, Sentinel<I> S2>
        requires Sentinel<S, I2>
        friend bool operator==(
            const common_iterator& x, const common_iterator<I2, S2>& y);
    
        template<class I2, Sentinel<I> S2>
        requires Sentinel<S, I2> && EqualityComparableWith<I, I2>
        friend bool operator==(
            const common_iterator& x, const common_iterator<I2, S2>& y);
        template<class I2, Sentinel<I> S2>
        requires Sentinel<S, I2>
        friend bool operator!=(
            const common_iterator& x, const common_iterator<I2, S2>& y);
    
        template<SizedSentinel<I> I2, SizedSentinel<I> S2>
        requires SizedSentinel<S, I2>
        friend iter_difference_t<I2> operator-(
            const common_iterator& x, const common_iterator<I2, S2>& y);
    
        friend iter_rvalue_reference_t<I> iter_move(const common_iterator& i)
        noexcept(noexcept(ranges::iter_move(declval<const I&>())))
        requires InputIterator<I>;
        template<IndirectlySwappable<I> I2, class S2>
        friend void iter_swap(const common_iterator& x, const common_iterator<I2, S2>& y)
        noexcept(noexcept(ranges::iter_swap(declval<const I&>(), declval<const I2&>())))
    
    private:
        variant<I, S> v_; // exposition only
    
    template<class I, class S>
    struct incrementable_traits<common_iterator<I, S>> {
        using difference_type = iter_difference_t<I>;
    
    };

    template<InputIterator I, class S>
    struct iterator_traits<common_iterator<I, S>> {
        using iterator_concept = see below;
        using iterator_category = see below;
    
}
using value_type = iter_value_t<I>;
using difference_type = iter_difference_t<I>;
using pointer = see below;
using reference = iter_reference_t<I>;
}

23.5.4.2 Associated types
[common.iter.types]
1 The nested typedef-names of the specialization of iterator_traits for common_iterator<I, S> are defined as follows.

(1.1) — iterator_concept denotes forward_iterator_tag if I models ForwardIterator; otherwise it denotes input_iterator_tag.

(1.2) — iterator_category denotes forward_iterator_tag if iterator_traits<I>::iterator_category models DerivedFrom<forward_iterator_tag>; otherwise it denotes input_iterator_tag.

(1.3) — If the expression a.operator->() is well-formed, where a is an lvalue of type const common_iterator<I, S>, then pointer denotes the type of that expression. Otherwise, pointer denotes void.

23.5.4.3 Constructors and conversions
[common.iter.const]
constexpr common_iterator(I i);
1 Effects: Initializes v_ as if by v_{in_place_type<I>, std::move(i)}.

constexpr common_iterator(S s);
2 Effects: Initializes v_ as if by v_{in_place_type<S>, std::move(s)}.

template<class I2, class S2>
requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S>
constexpr common_iterator(const common_iterator<I2, S2>& x);
3 Expects: x.v_.valueless_by_exception() is false.

4 Effects: Initializes v_ as if by v_{in_place_index<i>, get<i>(x.v_)}, where i is x.v_.index().

template<class I2, class S2>
requires ConvertibleTo<const I2&, I> && ConvertibleTo<const S2&, S> &&
Assignable<I&, const I2&> && Assignable<S&, const S2&>
common_iterator& operator=(const common_iterator<I2, S2>& x);
5 Expects: x.v_.valueless_by_exception() is false.

6 Effects: Equivalent to:

(6.1) — If v_.index() == x.v_.index(), then get<i>(v_) = get<i>(x.v_).

(6.2) — Otherwise, v_.emplace<i>(get<i>(x.v_)).

where i is x.v_.index().

7 Returns: *this

23.5.4.4 Accessors
[common.iter.access]
dcltype(auto) operator*();
dcltype(auto) operator*() const
requires dereferenceable<const I>;
1 Effects: holds_alternative<I>(v_).

2 Effects: Equivalent to: return *get<I>(v_);

dcltype(auto) operator->() const
requires see below;
3 The expression in the requires clause is equivalent to:

Readable<const I> &&
(requires(const I& i) { i.operator->(); }) ||
is_reference_v<iter_reference_t<I> ||
Constructible<iter_value_t<I>, iter_reference_t<I>>)

4 Expects: holds_alternative<I>(v_).

5 Effects:
(5.1) — If I is a pointer type or if the expression get<I>(v_).operator->() is well-formed, equivalent to: return get<I>(v_);
(5.2) — Otherwise, if iter_reference_t<I> is a reference type, equivalent to:
auto&& tmp = *get<I>(v_);
return addressof(tmp);
(5.3) — Otherwise, equivalent to: return proxy(*get<I>(v_)); where proxy is the exposition-only class:

class proxy {
  iter_value_t<I> keep_; 
proxy(iter_reference_t<I>&& x)
  : keep_(std::move(x)) {} 
public:
  const iter_value_t<I>* operator->() const {
    return addressof(keep_);
  }
};

23.5.4.5 Navigation [common.iter.nav]

common_iterator& operator++();
1 Expects: holds_alternative<I>(v_).
2 Effects: Equivalent to ++get<I>(v_).
3 Returns: *this.

dectype(auto) operator++(int);
4 Expects: holds_alternative<I>(v_).
5 Effects: If I models ForwardIterator, equivalent to:
common_iterator tmp = *this;
***this;
return tmp;
Otherwise, equivalent to: return get<I>(v_)++;

23.5.4.6 Comparisons [common.iter.cmp]

template<class I2, Sentinel<I> S2>
requires Sentinel<S, I2>
friend bool operator==(const common_iterator& x, const common_iterator<I2, S2>& y);
1 Expects: x.v_.valueless_by_exception() and y.v_.valueless_by_exception() are each false.
2 Returns: true if i == j, and otherwise get<i>(x.v_) == get<j>(y.v_), where i is x.v_.index() and j is y.v_.index().

template<class I2, Sentinel<I> S2>
requires Sentinel<S, I2> && EqualityComparableWith<I, I2>
friend bool operator==(const common_iterator& x, const common_iterator<I2, S2>& y);
3 Expects: 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().
template<class I2, Sentinel<I> S2>
    requires Sentinel<S, I2>
friend bool operator!=(const common_iterator& x, const common_iterator<I2, S2>& y);

Effects: Equivalent to: return !(x == y);

template<SizedSentinel<I> I2, SizedSentinel<I> S2>
    requires SizedSentinel<S, I2>
friend iter_difference_t<I2> operator-(const common_iterator& x, const common_iterator<I2, S2>& y);

Expects: 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 get<x>(x.v_) - get<j>(y.v_), where i is x.v_.index() and j is y.v_.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 InputIterator<I>;

1 Expects: holds_alternative<I>(v_).
2 Effects: Equivalent to: return ranges::iter_move(get<I>(i.v_));

template<IndirectlySwappable<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&>())))

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

Class template counted_iterator is an iterator adaptor with the same behavior as the underlying iterator except that it keeps track of the distance to the end of its range. It can be used together with default_sentinel in calls to generic algorithms to operate on a range of \( N \) elements starting at a given position without needing to know the end position a priori.

[Example:

```c++
list<string> s;
    // populate the list s with at least 10 strings
vector<string> v;
    // copies 10 strings into v:
ranges::copy(counted_iterator(s.begin(), 10), default_sentinel, back_inserter(v));
```
—end example]

Two values i1 and i2 of types counted_iterator<I1> and counted_iterator<I2> refer to elements of the same sequence if and only if next(i1.base(), i1.count()) and next(i2.base(), i2.count()) refer to the same (possibly past-the-end) element.

namespace std {
    template<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>
    requiresConvertibleTo<const I2&, I>
    constexpr counted_iterator(const counted_iterator<I2>& x);

template<class I2>
    requiresAssignable<I&, const I2&>
    constexpr counted_iterator& operator=(const counted_iterator<I2>& x);

cconstexpr I base() const;
cconstexpr iter_difference_t<I> count() const noexcept;
cconstexpr decltype(auto) operator*() require dereferenceable<const I>;

cconstexpr counted_iterator& operator++();
    decltype(auto) operator++(int);

cconstexpr counted_iterator operator++(int) require ForwardIterator<I>;

cconstexpr counted_iterator& operator--();
    requires BidirectionalIterator<I>;

cconstexpr counted_iterator operator--(int) require BidirectionalIterator<I>;

cconstexpr counted_iterator operator+(iter_difference_t<I> n) const require RandomAccessIterator<I>;
    friend constexpr counted_iterator operator+(iter_difference_t<I> n, const counted_iterator& x) require RandomAccessIterator<I>;

cconstexpr counted_iterator& operator+=(iter_difference_t<I> n) require RandomAccessIterator<I>;

cconstexpr counted_iterator operator-(iter_difference_t<I> n) const require RandomAccessIterator<I>;

template<Common<I> I2>
    friend constexpr iter_difference_t<I2> operator-(
        const counted_iterator& x, const counted_iterator<I2>& y);

template<Common<I> I2>
    friend constexpr bool operator==(const counted_iterator& x, const counted_iterator<I2>& y);

template<Common<I> I2>
    friend constexpr bool operator!=(const counted_iterator& x, const counted_iterator<I2>& y);

§ 23.5.6.1
template<Common<I> I2>
friend constexpr bool operator<(const counted_iterator& x, const counted_iterator<I2>& y);

template<Common<I> I2>
friend constexpr bool operator>(const counted_iterator& x, const counted_iterator<I2>& y);

template<Common<I> I2>
friend constexpr bool operator<=(const counted_iterator& x, const counted_iterator<I2>& y);

template<Common<I> I2>
friend constexpr bool 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 InputIterator<I>;

template<IndirectlySwappable<I> I2>
friend constexpr void iter_swap(const counted_iterator& x, const counted_iterator<I2>& y)
  noexcept(noexcept(ranges::iter_swap(x.current, y.current)));

private:
  I current = I(); // exposition only
  iter_difference_t<I> length = 0; // exposition only
};

template<class I>
struct incrementable_traits<counted_iterator<I>> {
  using difference_type = iter_difference_t<I>;
};

template<InputIterator I>
struct iterator_traits<counted_iterator<I>> : iterator_traits<I> {
  using pointer = void;
};

23.5.6.2 Constructors and conversions

constexpr counted_iterator(I i, iter_difference_t<I> n);
1

  \textbf{Expects:} n >= 0.

  \textbf{Effects:} Initializes \texttt{current} with \texttt{i} and \texttt{length} with \texttt{n}.

template<class I2>
requires ConvertibleTo<const I2&, I>
constexpr counted_iterator(const counted_iterator<I2>& x);
3

  \textbf{Effects:} Initializes \texttt{current} with \texttt{x.current} and \texttt{length} with \texttt{x.length}.

template<class I2>
requires Assignable<I&, const I2&>
constexpr counted_iterator& operator=(const counted_iterator<I2>& x);
4

  \textbf{Effects:} Assigns \texttt{x.current} to \texttt{current} and \texttt{x.length} to \texttt{length}.

  \textbf{Returns:} *this.

23.5.6.3 Accessors

constexpr I base() const;
1

  \textbf{Effects:} Equivalent to: return \texttt{current};

constexpr iter_difference_t<I> count() const noexcept;
2

  \textbf{Effects:} Equivalent to: return \texttt{length};

\section*{§ 23.5.6.3}
23.5.6.4 Element access

```cpp
constexpr decltype(auto) operator*();
constexpr decltype(auto) operator*() const
  requires dereferenceable<const I>;
```

**Effects:** Equivalent to: `return *current;`

```cpp
constexpr decltype(auto) operator[](iter_difference_t<I> n) const
  requires RandomAccessIterator<I>;
```

**Effects:** `n < length.`

**Expects:**

**Effects:** Equivalent to:

```cpp
return current[n];
```

23.5.6.5 Navigation

```cpp
constexpr counted_iterator& operator++();
```

**Expects:** `length > 0.`

**Effects:** Equivalent to:

```cpp
++current;
--length;
return *this;
```

```cpp
decltype(auto) operator++(int);
```

**Expects:** `length > 0.`

**Effects:** Equivalent to:

```cpp
try { return current++; } 
catch(...) { ++length; throw; }
```

```cpp
constexpr counted_iterator operator++(int)
  requires ForwardIterator<I>;
```

**Effects:** Equivalent to:

```cpp
counted_iterator tmp = *this;
++*this;
return tmp;
```

```cpp
constexpr counted_iterator& operator--();
```

**Effects:** Equivalent to:

```cpp
--current;
++length;
return *this;
```

```cpp
constexpr counted_iterator operator--(int)
  requires BidirectionalIterator<I>;
```

**Effects:** Equivalent to:

```cpp
counted_iterator tmp = *this;
--*this;
return tmp;
```

```cpp
constexpr counted_iterator operator+(iter_difference_t<I> n) const
  requires RandomAccessIterator<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 RandomAccessIterator<I>;
```

**Effects:** Equivalent to:

```cpp
return x + n;
```
constexpr counted_iterator& operator+=(iter_difference_t<I> n)
requires RandomAccessIterator<I>;

Expects: n <= length.
Effects: Equivalent to:
    current += n;
    length -= n;
    return *this;

constexpr counted_iterator operator-(iter_difference_t<I> n) const
requires RandomAccessIterator<I>;

Effects: Equivalent to: return counted_iterator(current - n, length + n);

template<Common<I> I2>
friend constexpr iter_difference_t<I2> operator-(
    const counted_iterator& x, const counted_iterator<I2>& y);

Expects: x and y refer to elements of the same sequence (23.5.6.1).
Effects: Equivalent to: return y.length - x.length;

friend constexpr iter_difference_t<I> operator-(
    const counted_iterator& x, default_sentinel_t);

Effects: Equivalent to: return -x.length;

friend constexpr iter_difference_t<I> operator-(
    default_sentinel_t, const counted_iterator& y);

Effects: Equivalent to: return y.length;

constexpr counted_iterator& operator-=(iter_difference_t<I> n)
requires RandomAccessIterator<I>;

Expects: -n <= length.
Effects: Equivalent to:
    current -= n;
    length += n;
    return *this;

23.5.6.6 Comparisons

template<Common<I> I2>
friend constexpr bool operator==(const counted_iterator& x, const counted_iterator<I2>& y);

Expects: 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);
friend constexpr bool operator==(default_sentinel_t, const counted_iterator& x);

Effects: Equivalent to: return x.length == 0;

template<Common<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 y);
friend constexpr bool operator!=(default_sentinel_t x, const counted_iterator& y);

Effects: Equivalent to: return !(x == y);
template<Common<I> I2>
friend constexpr bool operator<(const counted_iterator& x, const counted_iterator<I2>& y);

5 Expects: x and y refer to elements of the same sequence (23.5.6.1).

6 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]

template<Common<I> I2>
friend constexpr bool operator>(const counted_iterator& x, const counted_iterator<I2>& y);

8 Effects: Equivalent to: return y < x;

template<Common<I> I2>
friend constexpr bool operator<=(const counted_iterator& x, const counted_iterator<I2>& y);

9 Effects: Equivalent to: return !(y < x);

template<Common<I> I2>
friend constexpr bool operator>=(const counted_iterator& x, const counted_iterator<I2>& y);
10 Effects: Equivalent to: return !(x < y);

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 InputIterator<I>;
1 Effects: Equivalent to: return ranges::iter_move(i.current);

template<IndirectlySwappable<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)));
2 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]

1 Class unreachable_sentinel_t can be used with any WeaklyIncrementable type to denote the “upper bound” of an unbounded interval.

2 [Example:

```cpp
char* p;
// set p to point to a character buffer containing newlines
char* ml = find(p, unreachable_sentinel, '\n');
```

Provided a newline character really exists in the buffer, the use of unreachable_sentinel above potentially makes the call to find more efficient since the loop test against the sentinel does not require a conditional branch. —end example]

namespace std {
    struct unreachable_sentinel_t {
        template<WeaklyIncrementable I>
        friend constexpr bool operator==(unreachable_sentinel_t, const I&) noexcept;
        template<WeaklyIncrementable I>
        friend constexpr bool operator!=(unreachable_sentinel_t, const I&) noexcept;
        template<WeaklyIncrementable I>
        friend constexpr bool operator<(unreachable_sentinel_t, const I&) noexcept;
        template<WeaklyIncrementable I>
        friend constexpr bool operator!=(unreachable_sentinel_t, const I&) noexcept;
        template<WeaklyIncrementable I>
        friend constexpr bool operator>=(unreachable_sentinel_t, const I&) noexcept;
        template<WeaklyIncrementable I>
        friend constexpr bool operator!=(unreachable_sentinel_t, const I&) noexcept;
    }
}
23.5.7.2 Comparisons

```
};
}
```
private:
    basic_istream<charT,traits>* in_stream;  // exposition only
    T value;                                // exposition only
};

The type T shall meet the Cpp17DefaultConstructible, Cpp17CopyConstructible, and Cpp17CopyAssignable requirements.

### 23.6.1.1 Constructors and destructor

**constexpr istream_iterator();**

**constexpr istream_iterator(const istream_iterator& x) = default;**

**istream_iterator(const istream_iterator& x) = default;**

**~istream_iterator() = default;**

### 23.6.1.2 Operations

**const T& operator*() const;**

**const T* operator->() const;**

**istream_iterator& operator++();**

**istream_iterator operator++(int);**

**template<class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
                const istream_iterator<T,charT,traits,Distance>& y);**

```
§ 23.6.1.2 929
```
friend bool operator==(default_sentinel_t, const istream_iterator& i);
friend bool operator==(const istream_iterator& i, default_sentinel_t);

Returns: !i.in_stream.

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);
friend bool operator!=(default_sentinel_t x, const istream_iterator& y);
friend bool operator!=(const istream_iterator& x, default_sentinel_t y);

Returns: !(x == y)

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.

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 ostreambuf_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);

Effects: Initializes out_stream with addressof(s) and delim with nullptr.

ostream_iterator(ostream_type& s, const charT* delimiter);

Effects: Initializes out_stream with addressof(s) and delim with delimiter.

23.6.2.2 Operations

ostream_iterator& operator=(const T& value);

Effects: As if by:
    *out_stream << value;
    if (delim)
        *out_stream << delim;
return *this;

ostream_iterator& operator*();

Returns: *this.

ostream_iterator& operator++();

ostream_iterator& operator++(int);

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 end-of-range. All specializations of `istreambuf_iterator` shall have a trivial copy constructor, a `constexpr` default constructor, and a trivial destructor.

The result of `operator*()` on an end-of-stream iterator is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

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==(default_sentinel_t s, const istreambuf_iterator& i);
    friend bool operator==(const istreambuf_iterator& i, default_sentinel_t s);
    friend bool operator!=(default_sentinel_t a, const istreambuf_iterator& b);
    friend bool operator!=(const istreambuf_iterator& a, default_sentinel_t b);

  private:
    streambuf_type* sbuf_; // exposition only
  }

};

§ 23.6.3
23.6.3.1 Class istreambuf_iterator::proxy

Class istreambuf_iterator<charT,traits>::proxy is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class istreambuf_iterator<charT, traits>::proxy provides a temporary placeholder as the return value of the post-increment operator (operator++). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

namespace std {
    template<class charT, class traits>
    class istreambuf_iterator<charT, traits>::proxy { // exposition only
        charT keep_; 
        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.

constexpr istreambuf_iterator() noexcept;

effects: Initializes sbuf_ with nullptr.

constexpr istreambuf_iterator(default_sentinel_t) noexcept;

effects: Initializes sbuf_ with nullptr.

istreambuf_iterator(istream_type& s) noexcept;

effects: Initializes sbuf_ with s.rdbuf().

istreambuf_iterator(streambuf_type* s) noexcept;

effects: Initializes sbuf_ with s.

istreambuf_iterator(const proxy& p) noexcept;

effects: Initializes sbuf_ with p.sbuf_.

23.6.3.3 Operations

charT operator*() const

returns: The character obtained via the streambuf member sbuf_->sgetc().

istreambuf_iterator& operator++();

effects: As if by sbuf_->sbumpc().

Returns: *this.

proxy operator++(int);

returns: proxy(sbuf_->sbumpc(), sbuf_).

bool equal(const istreambuf_iterator& b) const;

returns: true if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what streambuf object they use.

template<class charT, class traits>

bool operator===(const istreambuf_iterator<charT,traits>& a,
                const istreambuf_iterator<charT,traits>& b);

returns: a.equal(b).

friend bool operator==(default_sentinel_t s, const istreambuf_iterator& i);

friend bool operator==(const istreambuf_iterator& i, default_sentinel_t s);

returns: i.equal(s).
template<class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
        const istreambuf_iterator<charT,traits>& b);
friend bool operator!=(default_sentinel_t a, const istreambuf_iterator& b);
friend bool operator!=(const istreambuf_iterator& a, default_sentinel_t b);

Returns: !a.equal(b).

23.6.4 Class template ostreambuf_iterator

The class template `ostreambuf_iterator` writes successive characters onto the output stream from which it was constructed.

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;

Effects: Initialize `sbuf_` with `s.rdbuf()`.

ostreambuf_iterator(streambuf_type* s) noexcept;

Effects: Initialize `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>`, `<deque>`, `<forward_list>`, `<list>`, `<regex>`, `<set>`, `<span>`, `<string>`, `<string_view>`, `<unordered_map>`, `<unordered_set>`, and `<vector>`. Each of these templates is a designated customization point (16.5.4.2.1).

**Template Definitions**

```cpp
template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());
Returns: c.begin().
```

```cpp
template<class C> constexpr auto end(C& c) -> decltype(c.end());
template<class C> constexpr auto end(const C& c) -> decltype(c.end());
Returns: c.end().
```

```cpp
template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;
Returns: array.
```

```cpp
template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;
Returns: array + N.
```

```cpp
template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
-> decltype(std::begin(c));
Returns: std::begin(c).
```

```cpp
template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));
Returns: std::end(c).
```

```cpp
template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());
template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());
Returns: c.rbegin().
```

```cpp
template<class C> constexpr auto rend(C& c) -> decltype(c.rend());
template<class C> constexpr auto rend(const C& c) -> decltype(c.rend());
Returns: c.rend().
```

```cpp
template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
Returns: reverse_iterator<T*>(array + N).
```

```cpp
template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
Returns: reverse_iterator<T*>(array).
```

```cpp
template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
Returns: reverse_iterator<const E*>(il.end()).
```

```cpp
template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
Returns: reverse_iterator<const E*>(il.begin()).
```

```cpp
template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
Returns: std::rbegin(c).
```
template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(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())>>;
Returns: 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().

§ 23.7 935
# 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 79.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</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 <initializer_list>
#include <iterator>

namespace std::ranges {
    inline namespace unspecified {
        // 24.3, range access
        inline constexpr unspecified begin = unspecified;
        inline constexpr unspecified end = unspecified;
        inline constexpr unspecified cbegin = unspecified;
        inline constexpr unspecified cend = unspecified;
        inline constexpr unspecified rbegin = unspecified;
        inline constexpr unspecified rend = unspecified;
        inline constexpr unspecified crbegin = unspecified;
        inline constexpr unspecified crend = unspecified;

        inline constexpr unspecified size = unspecified;
        inline constexpr unspecified empty = unspecified;
        inline constexpr unspecified data = unspecified;
        inline constexpr unspecified cdata = unspecified;
    } // 24.3, range access

    // 24.4.2, ranges
    template<class T>
    using iterator_t = decltype(ranges::begin(declval<T&>()));

    template<class T>
    using sentinel_t = decltype(ranges::end(declval<T&>()));

    template<class T>
    concept Range = see below;

    // 24.4.3, sized ranges
    template<class>
    inline constexpr bool disable_sized_range = false;

    template<class T>
    concept SizedRange = see below;
}
```

§ 24.2
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 OutputRange = see below;

template<class T>
concept InputRange = see below;

template<class T>
concept ForwardRange = see below;

template<class T>
concept BidirectionalRange = see below;

template<class T>
concept RandomAccessRange = see below;

template<class T>
concept ContiguousRange = see below;

template<class T>
concept CommonRange = see below;

template<class T>
concept ViewableRange = see below;

// 24.5.2, class template view_interface
template<class D>
requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface;

// 24.5.3, sub-ranges
enum class subrange_kind : bool { unsized, sized };

template<Iterator I, Sentinel<I> S = I, subrange_kind K = see below>
requires (K == subrange_kind::sized || SizedSentinel<S, I>)
class subrange;

// 24.5.4, dangling iterator handling
struct dangling;

template<Range R>
using safe_iterator_t = conditional_t<forwarding-range<R>, iterator_t<R>, dangling>;

template<Range R>
using safe_subrange_t = conditional_t<forwarding-range<R>, subrange_iterator_t<R>, dangling>;

// 24.6.1, empty view
template<class T>
requires is_object_v<T>
class empty_view;

namespace view {
    template<class T>
    inline constexpr empty_view<T> empty{};
}
template<CopyConstructible T>
    requires is_object_v<T>
    class single_view;

namespace view { inline constexpr unspecified single = unspecified; }

template<WeaklyIncrementable W, Semiregular Bound = unreachable_sentinel_t>
    requires weakly-equality-comparable-with<W, Bound>
    class iota_view;

namespace view { inline constexpr unspecified iota = unspecified; }

namespace view { inline constexpr unspecified all = unspecified; }

template<ViewableRange R>
    using all_view = decltype(view::all(declval<R>()));

template<Range R>
    requires is_object_v<R>
    class ref_view;

namespace view { inline constexpr unspecified filter = unspecified; }

template<InputRange V, IndirectUnaryPredicate<iterator_t<V>> Pred>
    requires View<V> && is_object_v<Pred>
    class filter_view;

namespace view { inline constexpr unspecified transform = unspecified; }

template<View> class take_view;

namespace view { inline constexpr unspecified take = unspecified; }

template<InputRange V>
    requires View<V> && InputRange<iter_reference_t<iterator_t<V>>> &&
    (is_reference_v<iter_reference_t<iterator_t<V>>> ||
     View<iter_value_t<iterator_t<V>>>)
    class join_view;

namespace view { inline constexpr unspecified join = unspecified; }

template<class R>
    concept tiny-range = see below; // exposition only

template<InputRange V, ForwardRange Pattern>
    requires View<V> && View<Pattern> &&
    IndirectlyComparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
    (ForwardRange<V> || tiny-range<Pattern>)
    class split_view;
namespace view { inline constexpr unspecified split = unspecified; }

// 24.7.9, counted view
namespace view { inline constexpr unspecified counted = unspecified; }

// 24.7.10, common view
template<View V>
  requires (!CommonRange<V>)
  class common_view;

namespace view { inline constexpr unspecified common = unspecified; }

// 24.7.11, reverse view
template<View V>
  requires BidirectionalRange<V>
  class reverse_view;

namespace view { inline constexpr unspecified reverse = unspecified; }

namespace std {
  namespace view = ranges::view;

  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; }
}

24.3 Range access [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> is included.

24.3.1 ranges::begin [range.access.begin]
1 The name ranges::begin denotes a customization point object (16.4.2.2.6). The expression ranges::begin(E) for some subexpression E is expression-equivalent to:

   § 24.3.2 939  

(1.1) — E + 0 if E is an lvalue of array type (6.7.2).
(1.2) — Otherwise, if E is an lvalue, decay-copy(E.begin()) if it is a valid expression and its type I models Iterator.
(1.3) — Otherwise, decay-copy(begin(E)) if it is a valid expression and its type I models Iterator with overload resolution performed in a context that includes the declarations:
    template<class T> void begin(T&&) = delete;
    template<class T> void begin(initializer_list<T>&&) = delete;

    and does not include a declaration of ranges::begin.  

(1.4) — Otherwise, ranges::begin(E) is ill-formed. [Note: This case can result in substitution failure when ranges::begin(E) appears in the immediate context of a template instantiation. — end note]

2 [Note: Whenever ranges::begin(E) is a valid expression, its type models Iterator. — end note]

24.3.2 ranges::end [range.access.end]
1 The name ranges::end denotes a customization point object (16.4.2.2.6). The expression ranges::end(E) for some subexpression E is expression-equivalent to:

§ 24.3.2
© ISO/IEC N4810

---

1.1. — If \( E \) is an lvalue of array type (6.7.2) \( T \).

1.2. — Otherwise, if \( E \) is an lvalue, \( \text{decay-copy}(E, \text{end}()) \) if it is a valid expression and its type \( S \) models Sentinel<decltype(ranges::begin(E))>.

1.3. — Otherwise, \( \text{decay-copy}(E) \) if it is a valid expression and its type \( S \) models Sentinel<decltype(ranges::begin(E))> with overload resolution performed in a context that includes the declarations:

\[
\begin{align*}
\text{template<class T> void end(T&&) = delete; \\
\text{template<class T> void end(initializer_list<T>&&) = delete; }
\end{align*}
\]

and does not include a declaration of \( \text{ranges::end} \).

1.4. — Otherwise, \( \text{ranges::end}(E) \) is ill-formed.  [Note: This case can result in substitution failure when \( \text{ranges::end}(E) \) appears in the immediate context of a template instantiation. — end note]

---

24.3.3 \text{ranges::cbegin} [range.access.cbegin]

1 The name \( \text{ranges::cbegin} \) denotes a customization point object (16.4.2.2.6). The expression \( \text{ranges::cbegin}(E) \) for some subexpression \( E \) of type \( T \) is expression-equivalent to:

1.1. — \( \text{ranges::begin}(\text{static_cast<const T&>(E)}) \) if \( E \) is an lvalue.

1.2. — Otherwise, \( \text{ranges::begin}(\text{static_cast<const T&&>(E)}) \).

2 [Note: Whenever \( \text{ranges::cbegin}(E) \) is a valid expression, its type models Iterator. — end note]

---

24.3.4 \text{ranges::cend} [range.access.cend]

1 The name \( \text{ranges::cend} \) denotes a customization point object (16.4.2.2.6). The expression \( \text{ranges::cend}(E) \) for some subexpression \( E \) of type \( T \) is expression-equivalent to:

1.1. — \( \text{ranges::end}(\text{static_cast<const T&>(E)}) \) if \( E \) is an lvalue.

1.2. — Otherwise, \( \text{ranges::end}(\text{static_cast<const T&&>(E)}) \).

2 [Note: Whenever \( \text{ranges::cend}(E) \) is a valid expression, the types \( S \) and \( I \) of \( \text{ranges::cend}(E) \) and \( \text{ranges::cbegin}(E) \) model Sentinel<\( S, I > \). — end note]

---

24.3.5 \text{ranges::rbegin} [range.access.rbeg]

1 The name \( \text{ranges::rbegin} \) denotes a customization point object (16.4.2.2.6). The expression \( \text{ranges::rbegin}(E) \) for some subexpression \( E \) is expression-equivalent to:

1.1. — If \( E \) is an lvalue, \( \text{decay-copy}(E, \text{rbegin}()) \) if it is a valid expression and its type \( I \) models Iterator.

1.2. — Otherwise, \( \text{decay-copy}(\text{rbegin}(E)) \) if it is a valid expression and its type \( I \) models Iterator with overload resolution performed in a context that includes the declaration:

\[
\text{template<class T> void rbegin(T&&) = delete;}
\]

and does not include a declaration of \( \text{ranges::rbegin} \).

1.3. — Otherwise, \( \text{make_reverse_iterator}(\text{ranges::end}(E)) \) if both \( \text{ranges::begin}(E) \) and \( \text{ranges::end}(E) \) are valid expressions of the same type \( T \) which models BidirectionalIterator (23.3.4.12).

1.4. — Otherwise, \( \text{ranges::rbegin}(E) \) is ill-formed.  [Note: This case can result in substitution failure when \( \text{ranges::rbegin}(E) \) appears in the immediate context of a template instantiation. — end note]

2 [Note: Whenever \( \text{ranges::rbegin}(E) \) is a valid expression, its type models Iterator. — end note]

---

24.3.6 \text{ranges::rend} [range.access.rend]

1 The name \( \text{ranges::rend} \) denotes a customization point object (16.4.2.2.6). The expression \( \text{ranges::rend}(E) \) for some subexpression \( E \) is expression-equivalent to:

1.1. — If \( E \) is an lvalue, \( \text{decay-copy}(E, \text{rend}()) \) if it is a valid expression and its type \( S \) models Sentinel<decltype(ranges::rbegin(E))>.

1.2. — Otherwise, \( \text{decay-copy}(\text{rend}(E)) \) if it is a valid expression and its type \( S \) models Sentinel<decltype(ranges::rbegin(E))> with overload resolution performed in a context that includes the declaration:
template<class T> void rend(T&&) = delete;

and does not include a declaration of ranges::rend.

(1.3) — Otherwise, make_reverse_iterator(ranges::begin(E)) if both ranges::begin(E) and ranges::end(E) are valid expressions of the same type I which models BidirectionalIterator (23.3.4.12).
(1.4) — Otherwise, ranges::rend(E) is ill-formed. [Note: This case can result in substitution failure when ranges::rend(E) appears in the immediate context of a template instantiation. — end note]

2 [Note: Whenever ranges::rend(E) is a valid expression, the types S and I of ranges::rend(E) and ranges::rbegin(E) model Sentinel<S, I>. — end note]

24.3.7 ranges::crbegin

The name ranges::crbegin denotes a customization point object (16.4.2.2.6). The expression ranges::crbegin(E) for some subexpression E of type T is expression-equivalent to:

(1.1) — ranges::rbegin(static_cast<const T&>(E)) if E is an lvalue.
(1.2) — Otherwise, ranges::rbegin(static_cast<const T&&>(E)).

2 [Note: Whenever ranges::crbegin(E) is a valid expression, its type models 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 some subexpression E of type T is expression-equivalent to:

(1.1) — ranges::rend(static_cast<const T&>(E)) if E is an lvalue.
(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<S, I>. — end note]

24.3.9 ranges::size

The name size denotes a customization point object (16.4.2.2.6). The expression ranges::size(E) for some subexpression E with type T is expression-equivalent to:

(1.1) — decay-copy(extent_v<T>) if T is an array type (6.7.2).
(1.2) — Otherwise, if disable_sized_range<remove_cv_t<T>> (24.4.3) is false:
(1.2.1) — decay-copy(size()) if it is a valid expression and its type I models Integral.
(1.2.2) — Otherwise, decay-copy(size()) if it is a valid expression and its type I models Integral with overload resolution performed in a context that includes the declaration:

```cpp
template<class T> void size(T&&) = delete;
```

and does not include a declaration of ranges::size.

(1.3) — Otherwise, (ranges::end(E) - ranges::begin(E)) if it is a valid expression and the types I and S of ranges::begin(E) and ranges::end(E) model SizedSentinel<S, I> (23.3.4.8) and ForwardIterator<I>. However, E is evaluated only once.
(1.4) — Otherwise, ranges::size(E) is ill-formed. [Note: This case can result in substitution failure when ranges::size(E) appears in the immediate context of a template instantiation. — end note]

2 [Note: Whenever ranges::size(E) is a valid expression, its type models Integral. — end note]

24.3.10 ranges::empty

The name empty denotes a customization point object (16.4.2.2.6). The expression ranges::empty(E) for some subexpression E is expression-equivalent to:

(1.1) — bool((E).empty()) if it is a valid expression.
(1.2) — Otherwise, (ranges::size(E) == 0) if it is a valid expression.
(1.3) — Otherwise, EQ, where EQ is bool(ranges::begin(E) == ranges::end(E)) except that E is only evaluated once, if EQ is a valid expression and the type of ranges::begin(E) models ForwardIterator.
24.3.11 ranges::data

The name data denotes a customization point object (16.4.2.6). The expression ranges::data(E) for some subexpression E is expression-equivalent to:

- If E is an lvalue, decay-copy(E.data()) if it is a valid expression of pointer to object type.
- Otherwise, if ranges::begin(E) is a valid expression whose type models ContiguousIterator, ranges::begin(E) == ranges::end(E) ? nullptr : addressof(*ranges::begin(E)) except that E is evaluated only once.

- Otherwise, ranges::data(E) is ill-formed. [Note: This case can result in substitution failure when ranges::data(E) appears in the immediate context of a template instantiation. — end note]

24.3.12 ranges::cdata

The name cdata denotes a customization point object (16.4.2.6). The expression ranges::cdata(E) for some subexpression E of type T is expression-equivalent to:

- ranges::data(static_cast<const T&>(E)) if E is an lvalue.
- Otherwise, ranges::data(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 Iterator (23.3.4.6). Calling ranges::end on a range returns an object whose type models Sentinel<S, I>, together with the type I of the object returned by ranges::begin, models View<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 SizedRange concept refines Range with the requirement that the number of elements in the range can be determined in constant time using the ranges::size function. The View concept specifies requirements on a Range type with constant-time copy and assign 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 RandomAccessIterator (23.3.4.13). (Contiguous, bidirectional, forward, input, and output ranges are defined similarly.) Viewable ranges can be converted to views.

24.4.2 Ranges

The Range concept defines the requirements of a type that allows iteration over its elements by providing an iterator and sentinel that denote the elements of the range.

```cpp
template<class T>
concept range-impl = // exposition only
    requires(T&& t) {  
        ranges::begin(std::forward<T>(t)); // sometimes equality-preserving (see below)
        ranges::end(std::forward<T>(t));
    };

template<class T>
concept Range = range-impl<T>;
```
template<class T>
concept forwarding-range = // exposition only
    Range<T> && range-impl<T>;

2 The required expressions ranges::begin(std::forward<T>(t)) and ranges::end(std::forward<T>(t)) of the \textit{range-impl} concept do not require implicit expression variations (18.2).

3 Given an expression \(E\) such that \texttt{decltype((E))} is \(T\), \(T\) models \textit{range-impl} only if

\begin{itemize}
\item \(\text{ranges::begin}(E)\) and \(\text{ranges::end}(E)\) denote a range (23.3.1),
\item both \(\text{ranges::begin}(E)\) and \(\text{ranges::end}(E)\) are amortized constant time and non-modifying, and
\item if the type of \(\text{ranges::begin}(E)\) models \texttt{ForwardIterator}, \(\text{ranges::begin}(E)\) is equality-preserving.
\end{itemize}

4 [Note: Equality preservation of both \(\text{ranges::begin}\) and \(\text{ranges::end}\) enables passing a \texttt{Range} whose iterator type models \texttt{ForwardIterator} to multiple algorithms and making multiple passes over the range by repeated calls to \(\text{ranges::begin}\) and \(\text{ranges::end}\). Since \(\text{ranges::begin}\) is not required to be equality-preserving when the return type does not model \texttt{ForwardIterator}, repeated calls might not return equal values or might not be well-defined; \(\text{ranges::begin}\) should be called at most once for such a range. —end note]

5 Given an expression \(E\) such that \texttt{decltype((E))} is \(T\) and an lvalue \(t\) that denotes the same object as \(E\), \(T\) models \textit{forwarding-range} only if

\begin{itemize}
\item \(\text{ranges::begin}(E)\) and \(\text{ranges::begin}(t)\) are expression-equivalent,
\item \(\text{ranges::end}(E)\) and \(\text{ranges::end}(t)\) are expression-equivalent, and
\item the validity of iterators obtained from the object denoted by \(E\) is not tied to the lifetime of that object.
\end{itemize}

6 [Note: Since the validity of iterators is not tied to the lifetime of an object whose type models \textit{forwarding-range}, a function can accept arguments of such a type by value and return iterators obtained from it without danger of dangling. —end note]

7 [Example: Specializations of class template \texttt{subrange} (24.5.3) model \textit{forwarding-range}. \texttt{subrange} provides non-member rvalue overloads of \texttt{begin} and \texttt{end} with the same semantics as its member lvalue overloads, and \texttt{subrange}'s iterators - since they are “borrowed” from some other range - do not have validity tied to the lifetime of a \texttt{subrange} object. —end example]

\section{24.4.3 Sized ranges} [range.sized]

1 The \texttt{SizedRange} concept specifies the requirements of a \texttt{Range} type that knows its size in constant time with the \texttt{size} function.

template<class T>
concept SizedRange =
    Range<T> &&
    !disable_sized_range<remove_cvref_t<T>> &&
    requires(T& t) { ranges::size(t); };

2 Given an lvalue \(t\) of type \texttt{remove_reference_t<T>}, \(T\) models \texttt{SizedRange} only if

\begin{itemize}
\item \(\text{ranges::size}(t)\) is \(O(1)\), does not modify \(t\), and is equal to \(\text{ranges::distance}(t)\), and
\item if \(\texttt{iterator_t<T>}\) models \texttt{ForwardIterator}, \(\text{ranges::size}(t)\) is well-defined regardless of the evaluation of \(\text{ranges::begin}(t)\). [Note: \(\text{ranges::size}(t)\) is otherwise not required to be well-defined after evaluating \(\text{ranges::begin}(t)\). For example, \(\text{ranges::size}(t)\) might be well-defined for a \texttt{SizedRange} whose iterator type does not model \texttt{ForwardIterator} only if evaluated before the first call to \(\text{ranges::begin}(t)\). —end note]
\end{itemize}

3 [Note: The complexity requirement for the evaluation of \(\text{ranges::size}\) is non-amortized, unlike the case for the complexity of the evaluations of \(\text{ranges::begin}\) and \(\text{ranges::end}\) in the \texttt{Range} concept. —end note]

4 [Note: \texttt{disable_sized_range} allows use of range types with the library that satisfy but do not in fact model \texttt{SizedRange}. —end note]
### 24.4.4 Views

The View concept specifies the requirements of a Range type that has constant time copy, move, and assignment operators; that is, the cost of these operations is not proportional to the number of elements in the View.

[Example: Examples of Views are:

(2.1) — A Range type that wraps a pair of iterators.
(2.2) — A Range type that holds its elements by shared_ptr and shares ownership with all its copies.
(2.3) — A Range type that generates its elements on demand.

Most containers (Clause 22) are not views since copying the container copies the elements, which cannot be done in constant time. — end example]

```c
template<class T>
inline constexpr bool enable_view = see below;
```

```c
template<class T>
concept View =
  Range<T> && Semiregular<T> && enable_view<T>;
```

Since the difference between Range and View is largely semantic, the two are differentiated with the help of enable_view.

For a type T, the default value of enable_view<T> is:

(4.1) — If DerivedFrom<T, view_base> is true, true.
(4.2) — Otherwise, if T is a specialization of class template initializer_list (17.10), set (22.4.6), multiset (22.4.7), unordered_set (22.5.6), unordered_multiset (22.5.7), or match_results (30.10), false.
(4.3) — Otherwise, if both T and const T model Range and iter_reference_t<iterator_t<T>> is not the same type as iter_reference_t<iterator_t<const T>>, false. [Note: Deep const-ness implies element ownership, whereas shallow const-ness implies reference semantics. — end note]
(4.4) — Otherwise, true.

Pursuant to 16.5.4.2.1, users may specialize enable_view to true for types which model View, and false for types which do not.

### 24.4.5 Other range refinements

The OutputRange concept specifies requirements of a Range type for which ranges::begin returns a model of OutputIterator (23.3.4.10). InputRange, ForwardRange, BidirectionalRange, and RandomAccessRange are defined similarly.

```c
template<class R, class T>
concept OutputRange =
  Range<R> && OutputIterator<iterator_t<R>, T>;
```

```c
template<class T>
concept InputRange =
  Range<T> && InputIterator<iterator_t<T>>;
```

```c
template<class T>
concept ForwardRange =
  InputRange<T> && ForwardIterator<iterator_t<T>>;
```

```c
template<class T>
concept BidirectionalRange =
  ForwardRange<T> && BidirectionalIterator<iterator_t<T>>;
```

```c
template<class T>
concept RandomAccessRange =
  BidirectionalRange<T> && RandomAccessIterator<iterator_t<T>>;
```
ContiguousRange additionally requires that the ranges::data customization point (24.3.11) is usable with the range.

```cpp
template<class T>
concept ContiguousRange =
    RandomAccessRange<T> && ContiguousIterator<iterator_t<T>> &&
    requires(T & t) {
        { ranges::data(t) } -> Same<add_pointer_t<iter_reference_t<iterator_t<T>>>; 
    };
```

The CommonRange 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 CommonRange. — end example]

```cpp
template<class T>
concept CommonRange =
    Range<T> && Same<iterator_t<T>, sentinel_t<T>>;
```

ViewableRange concept specifies the requirements of a Range type that can be converted to a View safely.

```cpp
template<class T>
concept ViewableRange =
    Range<T> && (forwarding-range<T> || View<decay_t<T>>);
```

## 24.5 Range utilities

The components in this subclause are general utilities for representing and manipulating ranges.

### 24.5.1 Helper concepts

Many of the types in this subclause are specified in terms of the following exposition-only concepts:

```cpp
template<class R>
concept simple-view = // exposition only
    View<R> && Range<const R> &&
    Same<iterator_t<R>, iterator_t<const R>> &&
    Same<sentinel_t<R>, sentinel_t<const R>>;
```

```cpp
template<InputIterator I>
concept has-arrow = // exposition only
    is_pointer_v<I> || requires(I i) { i.operator->(); };
```

```cpp
template<class T, class U>
concept not-same-as = // exposition only
    !Same<remove_cvref_t<T>, remove_cvref_t<U>>;
```

### 24.5.2 View interface

The class template `view_interface` is a helper for defining View-like types that offer a container-like interface. It is parameterized with the type that is derived from it.

```cpp
namespace std::ranges {
    template<Range R>
    struct range-common-iterator-impl { // exposition only
        using type = common_iterator<iterator_t<R>, sentinel_t<R>>;
    };
    template<CommonRange R>
    struct range-common-iterator-impl<ranges::common_iterator<iterator_t<R>, sentinel_t<R>>, R> { // exposition only
        using type = iterator_t<R>;
    };
    template<Range R>
    using range-common-iterator = // exposition only
        typename range-common-iterator-impl<R>::type;
}
```
```cpp
template<class D>
requires is_class_v<D> && Same<D, remove_cv_t<D>>
class view_interface : public view_base {

private:
  constexpr D& derived() noexcept {
    return static_cast<D&>(*this); // exposition only
  }
  constexpr const D& derived() const noexcept {
    return static_cast<const D&>(*this); // exposition only
  }

public:
  constexpr bool empty() requires ForwardRange<D> {
    return ranges::begin(derived()) == ranges::end(derived());
  }
  constexpr bool empty() const requires ForwardRange<const D> {
    return ranges::begin(derived()) == ranges::end(derived());
  }

  constexpr explicit operator bool() requires requires { ranges::empty(derived()); } {
    return !ranges::empty(derived());
  }
  constexpr explicit operator bool() const requires requires { ranges::empty(derived()); } {
    return !ranges::empty(derived());
  }

  constexpr auto data() requires ContiguousIterator<iterator_t<D>> {
    return ranges::empty(derived()) ? nullptr : addressof(*ranges::begin(derived()));
  }
  constexpr auto data() const requires Range<const D> && ContiguousIterator<iterator_t<const D>> {
    return ranges::empty(derived()) ? nullptr : addressof(*ranges::begin(derived()));
  }

  constexpr auto size() requires ForwardRange<D> && SizedSentinel<sentinel_t<D>, iterator_t<D>> {
    return ranges::end(derived()) - ranges::begin(derived());
  }
  constexpr auto size() const requires ForwardRange<const D> && SizedSentinel<sentinel_t<const D>, iterator_t<const D>> {
    return ranges::end(derived()) - ranges::begin(derived());
  }

  constexpr decltype(auto) front() requires ForwardRange<D>;  
  constexpr decltype(auto) front() const requires ForwardRange<const D>;
  constexpr decltype(auto) back() requires BidirectionalRange<D> && CommonRange<D>; 
  constexpr decltype(auto) back() const 
    requires BidirectionalRange<const D> && CommonRange<const D>;

  template<typename R = D>
  constexpr decltype(auto) operator[](iter_difference_t<iterator_t<R>> n) {
    return ranges::begin(derived())[n];
  }
  template<typename R = const D>
  constexpr decltype(auto) operator[](iter_difference_t<iterator_t<R>> n) const {
    return ranges::begin(derived())[n];
  }
};
```
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 `DerivedFrom<view_interface<D>>` and `View`.

### 24.5.2.1 Members

```cpp
constexpr decltype(auto) front() requires ForwardRange<D>;
constexpr decltype(auto) front() const requires ForwardRange<const D>;
```

1. **Expects:** `!empty()`.

2. **Effects:** Equivalent to: `return *ranges::begin(derived());`

```cpp
constexpr decltype(auto) back() requires BidirectionalRange<D> && CommonRange<D>;
constexpr decltype(auto) back() const
    requires BidirectionalRange<const D> && CommonRange<const D>;
```

1. **Expects:** `!empty()`.

2. **Effects:** Equivalent to: `return *ranges::prev(ranges::end(derived()));`

### 24.5.3 Sub-ranges

The `subrange` class template combines together an iterator and a sentinel into a single object that models the `View` concept. Additionally, it models the `SizedRange` concept when the final template parameter is `subrange_kind::sized`.

```cpp
namespace std::ranges {
    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 DerivedFrom<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) } -> const tuple_element_t<0, T>&;
            { get<1>(t) } -> const tuple_element_t<1, T>&;
        };

    template<class T, class U, class V> concept pair-like-convertible-to = // exposition only
        !Range<T> && pair-like<remove_reference_t<T>> &&
        requires(T&& t) {
            { get<0>(std::forward<T>(t)) } -> ConvertibleTo<U>;
            { get<1>(std::forward<T>(t)) } -> ConvertibleTo<V>;
        };

    template<class T, class U, class V> concept pair-like-convertible-from = // exposition only
        !Range<T> && pair-like<T> && Constructible<T, U, V>;<

    template<class T> concept iterator-sentinel-pair = // exposition only
        !Range<T> && pair-like<T> &&
        Sentinel<tuple_element_t<1, T>, tuple_element_t<0, T>>;

    template<Iterator I, Sentinel<I> S = I, subrange_kind K = SizedSentinel<I> ? subrange_kind::sized : subrange_kind::unsized>
    requires (K == subrange_kind::sized || !SizedSentinel<I>)
    class subrange : public view_interface<subrange<I, S, K>> {
        private:
            static constexpr bool StoreSize = // exposition only
                K == subrange_kind::sized && !SizedSentinel<I>;
            I begin_ = I(); // exposition only
            S end_ = S(); // exposition only
            iter_difference_t<I> size_ = 0; // exposition only; present only
```
public:
subrange() = default;

constexpr subrange(I i, S s) requires (!StoreSize);

constexpr subrange(I i, S s, iter_difference_t<I> n)
  requires (K == subrange_kind::sized);

template<not_same_as<subrange> R>
  requires forwarding_range<R> &&
  ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r) requires (!StoreSize || SizedRange<R>);

template<forwarding_range R>
  requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r, 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_to<PairLike, I, S>
constexpr subrange(PairLike&& r) requires (!StoreSize)
  : subrange{std::get<0>(std::forward<PairLike>(r)),
             std::get<1>(std::forward<PairLike>(r))}
{};

template<pair_like_convertible_to<I, S> PairLike>
constexpr subrange(PairLike&& r, iter_difference_t<I> n)
  requires (K == subrange_kind::sized)
  : subrange{std::get<0>(std::forward<PairLike>(r)),
             std::get<1>(std::forward<PairLike>(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;
constexpr S end() const;

constexpr bool empty() const;
constexpr iter_difference_t<I> size() const
  requires (K == subrange_kind::sized);

[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const;
[[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1)
  requires BidirectionalIterator<I>;
constexpr subrange& advance(iter_difference_t<I> n);

friend constexpr I begin(subrange&& r) { return r.begin(); }
friend constexpr S end(subrange&& r) { return r.end(); }

};

template<Iterator I, Sentinel<I> S>
subrange(I, S, iter_difference_t<I>) -> subrange<I, S, subrange_kind::sized>;

template<iterator_sentinel_pair P>
subrange(P) -> subrange<tuple_element_t<P>, tuple_element_t<I>, P>>;

template<iterator_sentinel_pair P>
subrange(P, iter_difference_t<tuple_element_t<P>, tuple_element_t<I>, P>>)
  ->
subrange<tuple_element_t<P>, tuple_element_t<I>, subrange_kind::sized>;
template<forwarding-range R>
subrange(R&&) ->
    subrange<iterator_t<R>, sentinel_t<R>,
        (SizedRange<R> || SizedSentinel<sentinel_t<R>, iterator_t<R>>)>
        subrange_kind::sized : subrange_kind::unsized;

template<forwarding-range R>
subrange(R&&, iter_difference_t<iterator_t<R>>) ->
    subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

template<size_t N, class I, class S, subrange_kind K>
requires (N < 2)
constexpr auto get(const subrange<I, S, K>& r);
}

namespace std {
    using ranges::get;
}

24.5.3.1 Constructors and conversions

constexpr subrange(I i, S s) requires (!StoreSize);
1    Expects: [i, s) is a valid range.
2    Effects: Initializes begin_ with i and end_ with s.

constexpr subrange(I i, S s, iter_difference_t<I> n)
requires (K == subrange_kind::sized);
3    Expects: [i, s) is a valid range, and n == ranges::distance(i, s).
4    Effects: Initializes begin_ with i and end_ with s. If StoreSize is true, initializes size_ with n.
5    [Note: Accepting the length of the range and storing it to later return from size() enables subrange
to model SizedRange even when it stores an iterator and sentinel that do not model SizedSentinel.
    —end note]

template<not_same_as<subrange> R>
requires forwarding-range<R> &&
   ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r) requires (!StoreSize || SizedRange<R>);
6    Effects: Equivalent to:
(6.1) — If StoreSize is true, subrange{ranges::begin(r), ranges::end(r), ranges::size(r)}.
(6.2) — Otherwise, subrange{ranges::begin(r), ranges::end(r)}.

template<not_same_as<subrange> PairLike>
    requires pair-like-convertible-from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
7    Effects: Equivalent to: return PairLike(begin_, end_);

24.5.3.2 Accessors

constexpr I begin() const;
1    Effects: Equivalent to: return begin_;

constexpr S end() const;
2    Effects: Equivalent to: return end_;

constexpr bool empty() const;
3    Effects: Equivalent to: return begin_ == end_;

constexpr iter_difference_t<I> size() const
requires (K == subrange_kind::sized);
4    Effects:
If StoreSize is true, equivalent to: return size_;  
Otherwise, equivalent to: return end_ - begin_;  

[[nodiscard]] constexpr subrange next(iter_difference_t<I> n = 1) const;  
Effects: Equivalent to:  
auto tmp = *this;  
tmp.advance(n);  
return tmp;  

[Note: If I does not model ForwardIterator, next can invalidate *this. — end note]  

[[nodiscard]] constexpr subrange prev(iter_difference_t<I> n = 1) const  
requires BidirectionalIterator<I>;  
Effects: Equivalent to:  
auto tmp = *this;  
tmp.advance(-n);  
return tmp;  

constexpr subrange& advance(iter_difference_t<I> n);  
Effects: Equivalent to:  
— If StoreSize is true,  
size_ -= n - ranges::advance(begin_, n, end_);  
return *this;  
— Otherwise,  
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);  
Effects: Equivalent to:  
if constexpr (N == 0)  
return r.begin();  
else  
return r.end();

24.5.4 Dangling iterator handling  

The tag type dangling is used together with the template aliases safe_iterator_t and safe_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 forwarding-range (24.4.2).

namespace std {  
struct dangling {  
constexpr dangling() noexcept = default;  
template<class... Args>  
constexpr dangling(Args&&...) noexcept { }  
};  
}  

[Example:  
vector<int> f();  
auto result1 = ranges::find(f(), 42); // #1  
static_assert(Same<decltype(result1), dangling>());  
auto vec = f();  
auto result2 = ranges::find(vec, 42); // #2  
static_assert(Same<decltype(result2), vector<int>::iterator>());  
auto result3 = ranges::find(subrange(vec), 42); // #3  
static_assert(Same<decltype(result3), vector<int>::iterator>());  
}
The call to `ranges::find` at #1 returns 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 `forwarding-range`. — end example

### 24.6 Range factories

This subclause defines *range factories*, which are utilities to create a View.

#### 24.6.1 Empty view

*empty_view* produces a View of no elements of a particular type.

*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`

```cpp
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 ptrdiff_t size() noexcept { return 0; }
      static constexpr bool empty() noexcept { return true; }

      friend constexpr T* begin(empty_view) noexcept { return nullptr; }
      friend constexpr T* end(empty_view) noexcept { return nullptr; }
  };
}
```
constexpr T* begin() noexcept;
constexpr const T* begin() const noexcept;
constexpr T* end() noexcept;
constexpr const T* end() const noexcept;
static constexpr ptrdiff_t size() noexcept;
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
};

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 ptrdiff_t size() noexcept;
Effects: Equivalent to: return 1;
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
Effects: Equivalent to: return value_.operator->();

24.6.2.3 view::single
The name view::single denotes a customization point object (16.4.2.2.6). For some subexpression E, the expression view::single(E) is expression-equivalent to single_view(E).
24.6.3 Iota view
24.6.3.1 Overview

iota_view generates a sequence of elements by repeatedly incrementing an initial value.

[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<WeaklyIncrementable W, Semiregular Bound = unreachable_sentinel_t>
requires weakly-equality-comparable-with<W, Bound>
class iota_view : public view_interface<iota_view<W, Bound>> {
  private:
  // 24.6.3.3, class iota_view::iterator
  struct iterator;  // exposition only
  // 24.6.3.4, class iota_view::sentinel
  struct sentinel;  // exposition only

  W value_ = W();  // exposition only
  Bound bound_ = Bound();  // exposition only

  public:
  iota_view() = default;
  constexpr explicit iota_view(W value);
  constexpr iota_view(type_identity_t<W> value,
                       type_identity_t<Bound> bound);

  constexpr iterator begin() const;
  constexpr sentinel end() const;
  constexpr iterator end() const requires Same<W, Bound>;

  constexpr auto size() const
  requires (Same<W, Bound> && Advanceable<W>) ||
  (Integral<W> && Integral<Bound>) ||
  SizedSentinel<Bound, W>
  {
    return bound_ - value_;  
  }
};

template<class W, class Bound>
requires (!Integral<W> || !Integral<Bound> || is_signed_v<W> == is_signed_v<Bound>)
iota_view(W, Bound) -> iota_view<W, Bound>;

1 The exposition-only Decrementable concept is equivalent to:

```
template<class I>
concept Decrementable =
  Incrementable<I> && requires(I i) {
    { --i } -> Same<I&>;
    { i-- } -> Same<I>;
  };
```

2 When an object is in the domain of both pre- and post-decrement, the object is said to be decrenable.

Let a and b be equal objects of type I. I models Decrementable only if:

(3.1) — If a and b are decrenable, then the following are all true:

(3.1.1) — addressof(--a) == addressof(a)
(3.1.2) — bool(a-- == b)
(3.1.3) — bool(((void)a--, a) == --b)
(3.1.4) — bool(++(--a) == b).

(3.2) — If a and b are increnable, then bool(--(++a) == b).

4 The exposition-only Advanceable concept is equivalent to:

```
template<class I>
concept Advanceable =
  Decrementable<I> && StrictTotallyOrdered<I> &&
  requires(I i, const I j, const iter_difference_t<I> n) {
    { i += n } -> Same<I&>;
    { i -= n } -> Same<I&>;
    { j + n } -> Same<I>;
    { n + j } -> Same<I>;
    { j - n } -> Same<I>;
    { j - j } -> Same<iter_difference_t<I>>;
  };
```

§ 24.6.3.2
Let \( a \) and \( b \) be objects of type \( I \) such that \( b \) is reachable from \( a \) after \( n \) applications of \( ++a \), for some value \( n \) of type \( \text{iter_difference}_t<I> \), and let \( D \) be \( \text{iter_difference}_t<I> \). \( I \) models \textit{Advanceable} only if

1. \( (a += n) \) is equal to \( b \).
2. \( \text{addressof}(a += n) \) is equal to \( \text{addressof}(a) \).
3. \( (a + n) \) is equal to \( (a += n) \).
4. For any two positive values \( x \) and \( y \) of type \( D \), if \( (a + D(x + y)) \) is well-defined, then \( (a + D(x + y)) \) is equal to \( ((a + x) + y) \).
5. \( (a + D(0)) \) is equal to \( a \).
6. If \( (a + D(n - 1)) \) is well-defined, then \( (a + n) \) is equal to \( ++(a + D(n - 1)) \).
7. \( (b += -n) \) is equal to \( a \).
8. \( (b -= n) \) is equal to \( a \).
9. \( \text{addressof}(b -= n) \) is equal to \( \text{addressof}(b) \).
10. \( (b - n) \) is equal to \( (b -= n) \).
11. \( (b - a) \) is equal to \( n \).
12. \( (a - b) \) is equal to \( -n \).
13. \( \text{bool}(a <= b) \) is true.

```cpp
constexpr explicit iota_view(W value);
```

**Expects:** Bound denotes \texttt{unreachable_sentinel_t} or \texttt{Bound()} is reachable from \texttt{value}.

**Effects:** Initializes \texttt{value_} with \texttt{value}.

```cpp
constexpr iota_view(type_identity_t<W> value, type_identity_t<Bound> bound);
```

**Expects:** Bound denotes \texttt{unreachable_sentinel_t} or \texttt{bound} is reachable from \texttt{value}.

**Effects:** Initializes \texttt{value_} with \texttt{value} and \texttt{bound_} with \texttt{bound}.

```cpp
constexpr iterator begin() const;
```

**Effects:** Equivalent to: return iterator\{value\};

```cpp
constexpr sentinel end() const;
```

**Effects:** Equivalent to: return sentinel\{bound\};

```cpp
constexpr iterator end() const requires Same<W, Bound>;
```

**Effects:** Equivalent to: return iterator\{bound\};

### 24.6.3.3 Class \texttt{iota_view::iterator} [range.iota.iterator]

```cpp
namespace std::ranges {
  template<class W, class Bound>
  struct iota_view<W, Bound>::iterator {
    private:
      W value_ = W(); // exposition only
    public:
      using iterator_category = see below;
      using value_type = W;
      using difference_type = iter_difference_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>;
  };
}
```

§ 24.6.3.3
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 EqualityComparable<W>
friend constexpr bool operator!=(const iterator& x, const iterator& y) requires EqualityComparable<W>
friend constexpr bool operator<(const iterator& x, const iterator& y) requires StrictTotallyOrdered<W>
friend constexpr bool operator>(const iterator& x, const iterator& y) requires StrictTotallyOrdered<W>
friend constexpr bool operator<=(const iterator& x, const iterator& y) requires StrictTotallyOrdered<W>
friend constexpr bool operator>=(const iterator& x, const iterator& y) requires StrictTotallyOrdered<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_category is defined as follows:

(1.1) — If W models Advanceable, then iterator_category is random_access_iterator_tag.
(1.2) — Otherwise, if W models Decrementable, then iterator_category is bidirectional_iterator_tag.
(1.3) — Otherwise, if W models Incrementable, then iterator_category is forward_iterator_tag.
(1.4) — Otherwise, iterator_category is input_iterator_tag.

2 [Note: Overloads for iter_move and iter_swap are omitted intentionally. — end note]

constexpr explicit iterator(W value);

3 Effects: Initializes value_ with value.
constexpr W operator*() const noexcept(is_nothrow_copy_constructible_v<W>);

4 Effects: Equivalent to: return value_;

5 [Note: The noexcept clause is needed by the default iter_move implementation. — end note]

constexpr iterator& operator++();

6 Effects: Equivalent to:
  ++value_;  return *this;
constexpr void operator++(int);

7 Effects: Equivalent to +++this.

constexpr iterator operator++(int) requires Incrementable<W>;

8 Effects: Equivalent to:
auto tmp = *this;
+++this;
return tmp;

constexpr iterator& operator--() requires Decrementable<W>;

Effects: Equivalent to:
---value_
return *this;

constexpr iterator operator--(int) requires Decrementable<W>;

Effects: Equivalent to:
auto tmp = *this;
---this;
return tmp;

constexpr iterator& operator+=(difference_type n)
requires Advanceable<W>;

Effects: Equivalent to:
value_ += n;
return *this;

constexpr iterator& operator-=(difference_type n)
requires Advanceable<W>;

Effects: Equivalent to:
value_ -= n;
return *this;

constexpr W operator[](difference_type n) const
requires Advanceable<W>;

Effects: Equivalent to: return value_ + n;

friend constexpr bool operator==(const iterator& x, const iterator& y)
requires EqualityComparable<W>;

Effects: Equivalent to: return x.value_ == y.value_

friend constexpr bool operator!=(const iterator& x, const iterator& y)
requires EqualityComparable<W>;

Effects: Equivalent to: return !(x == y);

friend constexpr bool operator<(const iterator& x, const iterator& y)
requires StrictTotallyOrdered<W>;

Effects: Equivalent to: return x.value_ < y.value_

friend constexpr bool operator<=(const iterator& x, const iterator& y)
requires StrictTotallyOrdered<W>;

Effects: Equivalent to: return !(y < x);

friend constexpr bool operator>(const iterator& x, const iterator& y)
requires StrictTotallyOrdered<W>;

Effects: Equivalent to: return !(x < y);

friend constexpr iterator operator+(iterator i, difference_type n)
requires Advanceable<W>;

Effects: Equivalent to: return iterator{i.value_ + n};

§ 24.6.3.3
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: return x.value_ - y.value_;

24.6.3.4 Class iota_view::sentinel

namespace std::ranges {
  template<class W, class Bound>
  struct iota_view<W, Bound>::sentinel {
    private:
      Bound bound_ = Bound(); // exposition only
    public:
      sentinel() = default;
      constexpr explicit sentinel(Bound bound);
      friend constexpr bool operator==(const iterator& x, const sentinel& y);
      friend constexpr bool operator==(const sentinel& x, const iterator& y);
      friend constexpr bool operator!=(const iterator& x, const sentinel& y);
      friend constexpr bool operator!=(const sentinel& x, const iterator& y);
    };
  }

  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 bool operator==(const sentinel& x, const iterator& y);

   Effects: Equivalent to: return y == x;

friend constexpr bool operator!=(const iterator& x, const sentinel& y);

   Effects: Equivalent to: return !(x == y);

friend constexpr bool operator!=(const sentinel& x, const iterator& y);

   Effects: Equivalent to: return !(y == x);

24.6.3.5 view::iota

The name view::iota denotes a customization point object (16.4.2.6). For some subexpressions E and F, the expressions view::iota(E) and view::iota(E, F) are expression-equivalent to iota_view{E} and iota_view{E, F}, respectively.

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::view.

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:
  vector<int> ints{0,1,2,3,4,5};

§ 24.7]
auto even = [](int i) { return 0 == i % 2; };
auto square = [](int i) { return i * i; };
for (int i : ints | view::filter(even) | view::transform(square)) {
    cout << i << ' '; // prints: 0 4 16
}
assert(ranges::equal(ints | view::filter(even), view::filter(ints, even));

—end example

24.7.1 Range adaptor objects

A range adaptor closure object is a unary function object that accepts a ViewableRange argument and
returns a View. For a range adaptor closure object C and an expression R such that decltype((R)) models
ViewableRange, the following expressions are equivalent and yield a View:
C(R)
R | C

Given an additional range adaptor closure object D, the expression C | D is well-formed and produces another
range adaptor closure object such that the following two expressions are equivalent:
R | C | D
R | (C | D)

A range adaptor object is a customization point object (16.4.2.2.6) that accepts a ViewableRange 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:
adapter(range, args...)
adapter(args...)(range)
range | adapter(args...)

In this case, adapter(args...) is a range adaptor closure object.

24.7.2 Semiregular wrapper

Many of the types in this subclause are specified in terms of an exposition-only class template semiregular.
semiregular<T> behaves exactly like optional<T> with the following differences:

(1.1) semiregular<T> constrains its type parameter T with CopyConstructible<T> && is_object_v<T>.
(1.2) If T models DefaultConstructible, the default constructor of semiregular<T> is equivalent to:

    constexpr semiregular() noexcept(is_nothrow_default_constructible_v<T>)
    : semiregular(in_place)
    {
    }

(1.3) If Assignable<T&, const T&> is not satisfied, the copy assignment operator is equivalent to:

    semiregular& operator=(const semiregular& that)
    noexcept(is_nothrow_copy_constructible_v<T>)
    {
        if (that) emplace(*that);
        else reset();
        return *this;
    }

(1.4) If Assignable<T&, T> is not satisfied, the move assignment operator is equivalent to:

    semiregular& operator=(semiregular&& that)
    noexcept(is_nothrow_move_constructible_v<T>)
    {
        if (that) emplace(std::move(*that));
        else reset();
        return *this;
    }

24.7.3 All view

view::all returns a View that includes all elements of its Range argument.
The name `view::all` denotes a range adaptor object (24.7.1). For some subexpression \( E \), the expression `view::all(E)` is expression-equivalent to:

1. **(2.1)** decay-copy\((E)\) if the decayed type of \( E \) models View.
2. **(2.2)** Otherwise, \( \text{ref_view}(E) \) if that expression is well-formed.
3. **(2.3)** Otherwise, subrange\((E)\).

### 24.7.3.1 Class template `ref_view`

`ref_view` is a View of the elements of some other Range.

```cpp
define std::ranges

template<Range R>
requires std::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 std::SizedRange<R>
        { return ranges::size(*r_); }

    constexpr auto data() const
        requires std::ContiguousRange<R>
        { return ranges::data(*r_); }

friend constexpr iterator_t<R> begin(ref_view r) { return r.begin(); }
friend constexpr sentinel_t<R> end(ref_view r) { return r.end(); }

};

template<not_same_as<ref_view> T>
requires see below
    constexpr ref_view(T&& t);

Remarks: Let \( \text{FUN} \) denote the exposition-only functions

```cpp
    void FUN(R&);
    void FUN(R&) = delete;
```

The expression in the requires-clause is equivalent to

```cpp
    \text{ConvertibleTo<T, R&> \& \ requires \{ FUN(declval<T>(t))\};
```

Effects: Initializes \( r_ \) with `addressof`\((static_cast<R&>(\text{std::forward<T>(t)})\))`.

### 24.7.4 Filter view

#### 24.7.4.1 Overview

`filter_view` presents a View of an underlying sequence without the elements that fail to satisfy a predicate.
2 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<InputRange V, IndirectUnaryPredicate<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<Pred> pred_; // exposition only

        // 24.7.4.3, class filter_view::iterator
        class iterator; // exposition only

        // 24.7.4.4, class filter_view::sentinel
        class sentinel; // exposition only

        public:
            filter_view() = default;
            constexpr filter_view(V base, Pred pred);
            template<InputRange R>
            requires ViewableRange<R> && Constructible<V, all_view<R>>
            constexpr filter_view(R&& r, Pred pred);
            constexpr V base() const;
            constexpr iterator begin();
            constexpr auto end() {
            if constexpr (CommonRange<V>)
                return iterator{*this, ranges::end(base_)};
            else
                return sentinel{*this};
            }
    };

template<class R, class Pred>
    filter_view(R&&, Pred) -> filter_view<all_view<R>, Pred>;
}

constexpr filter_view(V base, Pred pred);

Effects: Initializes base_ with std::move(base) and initializes pred_ with std::move(pred).

template<InputRange R>
    requires ViewableRange<R> && Constructible<V, all_view<R>>
    constexpr filter_view(R&& r, Pred pred);

Effects: Initializes base_ with view::all(std::forward<R>(r)) and initializes pred_ with std::move(pred).

constexpr V base() const;

Effects: Equivalent to: return base_;

constexpr iterator begin();

Expects: 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,
this function caches the result within the filter_view for use on subsequent calls.
24.7.4.3 Class `filter_view::iterator`

```cpp
namespace std::ranges {
  template<class V, class 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 = iter_value_t<iterator_t<V>>;
    using difference_type = iter_difference_t<iterator_t<V>>;

    iterator() = default;
    constexpr iterator(filter_view& parent, iterator_t<V> current);

    constexpr iterator_t<V> base() const;
    constexpr iter_reference_t<iterator_t<V>> operator*() const;
    constexpr iterator_t<V> operator->() const requires has-arrow<iterator_t<V>>;
    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int) requires BidirectionalRange<V>;
    constexpr iterator& operator--() requires BidirectionalRange<V>;
    constexpr iterator operator--(int) requires BidirectionalRange<V>;

    friend constexpr bool operator==(const iterator& x, const iterator& y)
      requires EqualityComparable<iterator_t<V>>;
    friend constexpr bool operator!=(const iterator& x, const iterator& y)
      requires EqualityComparable<iterator_t<V>>;

    friend constexpr iter_rvalue_reference_t<iterator_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 IndirectlySwappable<iterator_t<V>>;
  };

1 Modification of the element a `filter_view::iterator` denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

2 `iterator::iterator_concept` is defined as follows:

(2.1) If V models BidirectionalRange, then iterator_concept denotes bidirectional_iterator_tag.
(2.2) Otherwise, if V models ForwardRange, then iterator_concept denotes forward_iterator_tag.
(2.3) Otherwise, iterator_concept denotes input_iterator_tag.

3 `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 DerivedFrom<bidirectional_iterator_tag>, then iterator_category denotes bidirectional_iterator_tag.
(3.3) Otherwise, if C models DerivedFrom<forward_iterator_tag>, then iterator_category denotes forward_iterator_tag.
(3.4) Otherwise, iterator_category denotes input_iterator_tag.

4 Effects: Initializes current_ with current and parent_ with addressof(parent).
```
constexpr iterator_t<V> base() const;

Effects: Equivalent to: return current_

constexpr iter_reference_t<iterator_t<V>> operator*() const;

Effects: Equivalent to: return *current_

constexpr iterator_t<V> operator->() const

requires has_arrow<iterator_t<V>>;

Effects: Equivalent to: return current_

constexpr iterator& operator++();

Effects: Equivalent to:

current_ = ranges::find_if(++current_, ranges::end(parent_->base_), ref(*parent_->pred_));
return *this;

constexpr void operator++(int);

Effects: Equivalent to +++this.

constexpr iterator operator++(int) requires ForwardRange<V>;

Effects: Equivalent to:

auto tmp = *this;
+++this;
return tmp;

constexpr iterator& operator--() requires BidirectionalRange<V>;

Effects: Equivalent to:

do
--current_;
while (!invoke(*parent_->pred_, *current_));
return *this;

constexpr iterator operator--(int) requires BidirectionalRange<V>;

Effects: Equivalent to:

auto tmp = *this;
--*this;
return tmp;

friend constexpr bool operator==(const iterator& x, const iterator& y)
requires EqualityComparable<iterator_t<V>>;

Effects: Equivalent to: return x.current_ == y.current_

friend constexpr bool operator!=(const iterator& x, const iterator& y)
requires EqualityComparable<iterator_t<V>>;

Effects: Equivalent to: !(x == y);

friend constexpr iter_rvalue_reference_t<iterator_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 IndirectlySwappable<iterator_t<V>>;

Effects: Equivalent to: ranges::iter_swap(x.current_, y.current_).

24.7.4.4 Class filter_view::sentinel [range.filter.sentinel]

namespace std::ranges {
  template<class V, class Pred>
  class filter_view<V, Pred>::sentinel {
    private:

§ 24.7.4.4
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);
friend constexpr bool operator==(const sentinel& x, const iterator& y);
friend constexpr bool operator!=(const iterator& x, const sentinel& y);
friend constexpr bool operator!=(const sentinel& x, const iterator& y);
};

constexpr explicit sentinel(filter_view& parent);

Effects: Initializes end_ with ranges::end(parent).

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_;  

friend constexpr bool operator==(const sentinel& x, const iterator& y);

Effects: Equivalent to: return y == x; 

friend constexpr bool operator!=(const iterator& x, const sentinel& y);

Effects: Equivalent to: return !(x == y); 

friend constexpr bool operator!=(const sentinel& x, const iterator& y);

Effects: Equivalent to: return !(y == x); 

24.7.4.5 view::filter  

The name view::filter denotes a range adaptor object (24.7.1). For some subexpressions E and P, the expression view::filter(E, P) is expression-equivalent to filter_view{E, P}.

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.

[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<InputRange V, CopyConstructible F>
    requires View<V> && is_object_v<F> &&
    RegularInvocable<F&, iter_reference_t<iterator_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

§ 24.7.5.2 963
V base_ = V(); // exposition only
semiregular<F> fun_; // exposition only

public:
transform_view() = default;
constexpr transform_view(V base, F fun);
template<InputRange R>
  requires ViewableRange<R> && Constructible<V, all_view<R>>
constexpr transform_view(R&& r, F fun);

constexpr V base() const;
constexpr iterator<false> begin();
constexpr iterator<true> begin() const
  requires Range<const V> &&
  RegularInvocable<const F&, iter_reference_t<iterator_t<const V>>>;
constexpr sentinel<false> end();
constexpr iterator<false> end() requires CommonRange<V>
constexpr sentinel<true> end() const
  requires CommonRange<const V> &&
  RegularInvocable<const F&, iter_reference_t<iterator_t<const V>>>;
constexpr auto size() requires SizedRange<const V>
{ return ranges::size(base_); }
constexpr auto size() const requires SizedRange<V>{ return ranges::size(base_); }

};
template<class R, class F>
transfrom_view(R&&, F) -> transform_view<all_view<R>, F>;

Effects: Initializes base_ with std::move(base) and fun_ with std::move(fun).

template<InputRange R>
  requires ViewableRange<R> && Constructible<V, all_view<R>>
constexpr transform_view(R&& r, F fun);
Effects: Initializes base_ with view::all(std::forward<R>(r)) and fun_ with std::move(fun).

Effects: Equivalent to: return base_;

Effects: Equivalent to:
  return iterator<false>(*this, ranges::begin(base_));

Effects: Equivalent to:
  return iterator<true>(*this, ranges::begin(base_));

Effects: Equivalent to:
  return iterator<false>({ranges::end(base_)});
constexpr iterator<false> end() requires CommonRange<V>;

Effects: Equivalent to:
        return iterator<false>(*this, ranges::end(base_));

cconstexpr sentinel<true> end() const
requires Range<const V> &
        RegularInvocable<const F&, iter_reference_t<iterator_t<const V>>>;

Effects: Equivalent to:
        return sentinel<true>(ranges::end(base_));

cconstexpr iterator<true> end() const
requires CommonRange<const V> &
        RegularInvocable<const F&, iter_reference_t<iterator_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<class V, class F>
    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&, iter_reference_t<iterator_t<Base>>>>;
        using difference_type = iter_difference_t<iterator_t<Base>>;

        iterator() = default;
        constexpr iterator(Parent& parent, iterator_t<Base> current);
        constexpr iterator(iterator<!Const> i)
            requires Const & ConvertibleTo<iterator_t<V>, iterator_t<Base>>;

        constexpr iterator_t<Base> base() const;
        constexpr decltype(auto) operator*() const
            { return invoke(*parent_->fun_, *current_); }

        constexpr iterator& operator++();
        constexpr void operator++(int);
        constexpr iterator operator++(int) requires ForwardRange<Base>;

        constexpr iterator& operator--() requires BidirectionalRange<Base>;
        constexpr iterator operator--(int) requires BidirectionalRange<Base>;

        constexpr iterator& operator+=(difference_type n)
            requires RandomAccessRange<Base>;
        constexpr iterator& operator-=(difference_type n)
            requires RandomAccessRange<Base>;
        constexpr decltype(auto) operator[](difference_type n) const
            requires RandomAccessRange<Base>
            { return invoke(*parent_->fun_, current_[n]); }

        { return invoke(*parent_->fun_, current_[n]); }

§ 24.7.5.3
friend constexpr bool operator==(const iterator& x, const iterator& y)
   requires EqualityComparable<iterator_t<Base>>;
friend constexpr bool operator!=(const iterator& x, const iterator& y)
   requires EqualityComparable<iterator_t<Base>>;
friend constexpr bool operator<(const iterator& x, const iterator& y)
   requires RandomAccessRange<Base>;
friend constexpr bool operator>(const iterator& x, const iterator& y)
   requires RandomAccessRange<Base>;
friend constexpr bool operator<=(const iterator& x, const iterator& y)
   requires RandomAccessRange<Base>;
friend constexpr bool operator>=(const iterator& x, const iterator& y)
   requires RandomAccessRange<Base>;
friend constexpr iterator operator+(iterator i, difference_type n)
   requires RandomAccessRange<Base>;
friend constexpr iterator operator+(difference_type n, iterator i)
   requires RandomAccessRange<Base>;
friend constexpr iterator operator-(iterator i, difference_type n)
   requires RandomAccessRange<Base>;
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
   requires RandomAccessRange<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 IndirectlySwappable<iterator_t<Base>>;
};

1 iterator::iterator_concept is defined as follows:
   — If V models RandomAccessRange, then iterator_concept denotes random_access_iterator_tag.
   — Otherwise, if V models BidirectionalRange, then iterator_concept denotes bidirectional_iterator_tag.
   — Otherwise, if V models ForwardRange, then iterator_concept denotes forward_iterator_tag.
   — Otherwise, iterator_concept denotes input_iterator_tag.

2 Let C denote the type iterator_traits<iterator_t<Base>>::iterator_category. If C models Derived
   From<contiguous_iterator_tag>, then iterator_category denotes random_access_iterator_tag; oth-
   erwise, iterator_category denotes C.

constexpr iterator(Parent& parent, iterator_t<Base> current);

   — Effects: Initializes current_ with current and parent_ with addressof(parent).

constexpr iterator(iterator<!Const> i)
   requires Const && ConvertibleTo<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;

   — Effects: Equivalent to: return current_.

constexpr iterator& operator++();

   — Effects: Equivalent to:
constexpr void operator++(int);

**Effects:** Equivalent to `++current_`.

constexpr iterator operator++(int) requires ForwardRange<Base>;

**Effects:** Equivalent to:

```c++
    auto tmp = *this;
    ++*this;
    return tmp;
```

constexpr iterator& operator--() requires BidirectionalRange<Base>;

**Effects:** Equivalent to:

```c++
    --current_;
    return *this;
```

constexpr iterator operator--(int) requires BidirectionalRange<Base>;

**Effects:** Equivalent to:

```c++
    auto tmp = *this;
    --*this;
    return tmp;
```

constexpr iterator& operator+=(difference_type n) requires RandomAccessRange<Base>;

**Effects:** Equivalent to:

```c++
    current_ += n;
    return *this;
```

constexpr iterator& operator-=(difference_type n) requires RandomAccessRange<Base>;

**Effects:** Equivalent to:

```c++
    current_ -= n;
    return *this;
```

friend constexpr bool operator==(const iterator& x, const iterator& y) requires EqualityComparable<iterator_t<Base>>;

**Effects:** Equivalent to:

```c++
    return x.current_ == y.current_; 
```

friend constexpr bool operator!=(const iterator& x, const iterator& y) requires EqualityComparable<iterator_t<Base>>;

**Effects:** Equivalent to:

```c++
    return !(x == y);
```

friend constexpr bool operator<(const iterator& x, const iterator& y) requires RandomAccessRange<Base>;

**Effects:** Equivalent to:

```c++
    return x.current_ < y.current_; 
```

friend constexpr bool operator!=(const iterator& x, const iterator& y) requires RandomAccessRange<Base>;

**Effects:** Equivalent to:

```c++
    return y < x;
```

friend constexpr bool operator<=(const iterator& x, const iterator& y) requires RandomAccessRange<Base>;

**Effects:** Equivalent to:

```c++
    return !(y < x);
```

friend constexpr bool operator>=(const iterator& x, const iterator& y) requires RandomAccessRange<Base>;

**Effects:** Equivalent to:

```c++
    return !(x < y);
```
friend constexpr iterator operator+(iterator i, difference_type n)
  requires RandomAccessRange<Base>;
friend constexpr iterator operator+(difference_type n, iterator i)
  requires RandomAccessRange<Base>;
19  
  Effects: Equivalent to: return iterator{*i.parent_, i.current_ + n};
friend constexpr iterator operator-(iterator i, difference_type n)
  requires RandomAccessRange<Base>;
20  
  Effects: Equivalent to: return iterator{*i.parent_, i.current_ - n};
friend constexpr difference_type operator-(const iterator& x, const iterator& y)
  requires RandomAccessRange<Base>;
21  
  Effects: Equivalent to: return x.current_ - y.current_;
friend constexpr void iter_swap(const iterator& x, const iterator& y)
  requires IndirectlySwappable<iterator_t<Base>>;
22  
  Effects: Equivalent to ranges::iter_swap(x.current_, y.current_).

24.7.5.4 Class template transform_view::sentinel

namespace std::ranges {
  template<class V, class F>
  template<bool Const>
  class transform_view<V, F>::sentinel {
private:
    using Parent = conditional_t<Const, const transform_view, transform_view>;
    using Base = conditional_t<Const, const V, V>;
    sentinel_t<Base> end_ = sentinel_t<Base>();
public:
  sentinel() = default;
  constexpr explicit sentinel(sentinel_t<Base> end);
  constexpr sentinel(sentinel<!Const> i)
    requires Const && ConvertibleTo<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 bool operator==(const sentinel& x, const iterator<Const>& y);
  friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);
  friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);
  friend constexpr iter_difference_t<iterator_t<Base>>
    operator-(const iterator<Const>& x, const sentinel& y)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;
  friend constexpr iter_difference_t<iterator_t<Base>>
    operator-(const sentinel& y, const iterator<Const>& x)
    requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;

  };
}

constexpr explicit sentinel(sentinel_t<Base> end);
1  
  Effects: Initializes end_ with end.

constexpr sentinel(sentinel<!Const> i)
  requires Const && ConvertibleTo<sentinel_t<V>, sentinel_t<Base>>;
2  
  Effects: Initializes end_ with std::move(i.end_).

constexpr sentinel_t<Base> base() const;
3  
  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 bool operator==(const sentinel& x, const iterator<Const>& y);

Effects: Equivalent to: return y == x; 

friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);

Effects: Equivalent to: return !(x == y); 

friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);

Effects: Equivalent to: return !(y == x); 

friend constexpr iter_difference_t<iterator_t<Base>>
operator-(const iterator<Const>& x, const sentinel& y)
requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;

Effects: Equivalent to: return x.current_ - y.end_; 

friend constexpr iter_difference_t<iterator_t<Base>>
operator-(const sentinel& y, const iterator<Const>& x)
requires SizedSentinel<sentinel_t<Base>, iterator_t<Base>>;

Effects: Equivalent to: return x.end_ - y.current_; 

24.7.5.5 view::transform [range.transform.adaptor]
The name view::transform denotes a range adaptor object (24.7.1). For some subexpressions E and F, the expression view::transform(E, F) is expression-equivalent to transform_view{E, F}. 

24.7.6 Take view [range.take]
24.7.6.1 Overview [range.take.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. 

[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 [range.take.view]
namespace std::ranges {
template<View V>
class take_view : public view_interface<take_view<V>> {
    private:
        V base_ = V(); // exposition only
        iter_difference_t<iterator_t<V>> count_ = 0; // exposition only
        // 24.7.6.3, class template take_view::sentinel
template<bool> struct take_view::sentinel;
    public:
        take_view() = default;
        constexpr take_view(V base, iter_difference_t<iterator_t<V>> count);
        template<ViewableRange R>
            requires Constructible<V, all_view<R>>
        constexpr take_view(R&& r, iter_difference_t<iterator_t<V>> count);
        constexpr V base() const; 
        constexpr auto begin() requires (!simple_view<V>) {
            if constexpr (SizedRange<V>) {
                if constexpr (RandomAccessRange<V>)
                    return ranges::begin(base_); 

§ 24.7.6.2
690
constexpr auto begin() const requires Range<const V> {
    if constexpr (SizedRange<const V>) {
        if constexpr (RandomAccessRange<const V>)
            return ranges::begin(base_);
        else
            return counted_iterator{ranges::begin(base_), count_};
    } else
        return counted_iterator{ranges::begin(base_), size()};
}

constexpr auto end() requires (!simple_view<V>) {
    if constexpr (SizedRange<V>) {
        if constexpr (RandomAccessRange<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 (SizedRange<const V>) {
        if constexpr (RandomAccessRange<const V>)
            return ranges::begin(base_) + size();
        else
            return default_sentinel;
    } else
        return sentinel<true>(ranges::end(base_));
}

constexpr auto size() requires SizedRange<V> {
    auto n = ranges::size(base_);
    return ranges::min(n, static_cast<decltype(n)>(count_));
}

constexpr auto size() const requires SizedRange<const V> {
    auto n = ranges::size(base_);
    return ranges::min(n, static_cast<decltype(n)>(count_));
};

template<Range R>
    take_view(R&&, iter_difference_t<iterator_t<R>>)
    -> take_view<all_view<R>>;

constexpr take_view(V base, iter_difference_t<iterator_t<V>> count);

 Effect: Initializes base_ with std::move(base) and count_ with count.

template<ViewableRange R>
    requires Constructible<V, all_view<R>>
    constexpr take_view(R&& r, iter_difference_t<iterator_t<V>> count);

 Effect: Initializes base_ with view::all(std::forward<R>(r)) and count_ with count.

constexpr V base() const;

 Effect: Equivalent to: return base_;
24.7.6.3 Class template take_view::sentinel

namespace std::ranges {
    template<class V>
    template<bool Const>
    class take_view<V>::sentinel {
    private:
        using Base = conditional_t<Const, const V, V>; // exposition only
        using CI = counted_iterator<iterator_t<Base>>; // exposition only
        sentinel_t<Base> end_ = sentinel_t<Base>(); // exposition only
    public:
        sentinel() = default;
        constexpr explicit sentinel(sentinel_t<Base> end);
        constexpr sentinel(sentinel<!Const> s)
            requires Const && ConvertibleTo<sentinel_t<V>, sentinel_t<Base>>;
        constexpr sentinel_t<Base> base() const;
        friend constexpr bool operator==(const sentinel& x, const CI& y);
        friend constexpr bool operator==(const CI& y, const sentinel& x);
        friend constexpr bool operator!=(const sentinel& x, const CI& y);
        friend constexpr bool operator!=(const CI& y, const sentinel& x);
    }
}

constexpr explicit sentinel(sentinel_t<Base> end);
Effects: Initializes end_ with end.

constexpr sentinel(sentinel<!Const> s)
    requires Const && ConvertibleTo<sentinel_t<V>, sentinel_t<Base>>;
Effects: Initializes end_ with std::move(s.end_).

constexpr sentinel_t<Base> base() const;
Effects: Equivalent to: return end_;

friend constexpr bool operator==(const sentinel& x, const CI& y);
friend constexpr bool operator==(const CI& y, const sentinel& x);
Effects: Equivalent to: return y.count() == 0 || y.base() == x.end_;

friend constexpr bool operator!=(const sentinel& x, const CI& y);
friend constexpr bool operator!=(const CI& y, const sentinel& x);
Effects: Equivalent to: return !(x == y);

24.7.6.4 view::take
Effects: Moves ranges

The name view::take denotes a range adaptor object (24.7.1). For some subexpressions E and F, the expression view::take(E, F) is expression-equivalent to take_view{E, F}.

24.7.7 Join view
24.7.7.1 Overview

join_view flattens a View of ranges into a View.

Example:

vector<string> ss("hello", " ", "world", "!");
join_view greeting(ss);
for (char ch : greeting)
    cout << ch; // prints: hello world!
— end example]
24.7.7.2 Class template join_view

namespace std::ranges {
    template<InputRange V>
    requires View<V> && InputRange<iter_reference_t<iterator_t<V>>> &&
        (is_reference_v<iter_reference_t<iterator_t<V>>> ||
        View<iter_value_t<iterator_t<V>>>)
    class join_view : public view_interface<join_view<V>> {
        private:
            using InnerRng = // exposition only
                iter_reference_t<iterator_t<V>>;
        // 24.7.7.3, class template join_view::iterator
        template<bool Const>
            struct iterator;
        // exposition only
        // 24.7.7.4, class template join_view::sentinel
        template<bool Const>
            struct sentinel;
        // exposition only
        V base_ = V(); // exposition only
        all_view<InnerRng> inner_ = // exposition only, present only when !is_reference_v<InnerRng>
            all_view<InnerRng>();
        public:
        join_view() = default;
        constexpr explicit join_view(V base);
        template<InputRange R>
            requires ViewableRange<R> && Constructible<V, all_view<R>>
        constexpr explicit join_view(R&& r);
        constexpr auto begin() {
            return iterator<simple_view<V>>{*this, ranges::begin(base_)};
        }
        constexpr auto begin() const
            requires InputRange<const V> &&
                is_reference_v<iter_reference_t<iterator_t<const V>>> {
            return iterator<true>{*this, ranges::begin(base_)};
        }
        constexpr auto end() {
            if constexpr (ForwardRange<V> &&
                is_reference_v<InnerRng> && ForwardRange<InnerRng> &&
                CommonRange<V> && CommonRange<InnerRng>)
            return iterator<simple_view<V>>{*this, ranges::end(base_)};
            else
                return sentinel<true>{*this};
        }
        constexpr auto end() const
            requires InputRange<const V> &&
                is_reference_v<iter_reference_t<iterator_t<const V>>> {
            if constexpr (ForwardRange<const V> &&
                is_reference_v<iter_reference_t<iterator_t<const V>>> &&
                ForwardRange<iter_reference_t<iterator_t<const V>>> &&
                CommonRange<const V> &&
                CommonRange<iter_reference_t<iterator_t<const V>>>)
                return iterator<true>{*this, ranges::end(base_)};
            else
                return sentinel<true>{*this};

            template<class R>
                explicit join_view(R&&) -> join_view<all_view<R>>;
    }

§ 24.7.7.2 972
constexpr explicit join_view(V base);

Effects: Initializes base_ with std::move(base).

template<InputRange R>
requires ViewableRange<R> && Constructible<V, all_view<R>>
constexpr explicit join_view(R&& r);

Effects: Initializes base_ with view::all(std::forward<R>(r)).

24.7.7.3 Class template join_view::iterator

namespace std::ranges {
  template<class 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<iter_reference_t<iterator_t<Base>>>
      
    iterator_t<Base> outer_ = iterator_t<Base>(); // exposition only
    iterator_t<iter_reference_t<iterator_t<Base>>> inner_; // exposition only
    Parent* parent_ = nullptr; // exposition only

    constexpr void satisfy(); // exposition only

    public:

    using iterator_concept = see below;
    using iterator_category = see below;
    using value_type = iter_value_t<iterator_t<iter_reference_t<iterator_t<Base>>>;
    using difference_type = see below;

    iterator() = default;
    constexpr iterator(Parent& parent, iterator_t<V> outer);
    constexpr iterator(iterator<!Const> i)
      requires Const && ConvertibleTo<iterator_t<V>, iterator_t<Base>> &&
        ConvertibleTo<iterator_t<InnerRng>,
          iterator_t<iter_reference_t<iterator_t<Base>>>;

    constexpr decltype(auto) operator*() const { return *inner_; }
    constexpr iterator_t<Base> operator->() const
      requires has-arrow<iterator_t<Base>>;

    constexpr iterator& operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int)
      requires ref_is_glvalue && ForwardRange<Base> &&
        ForwardRange<iter_reference_t<iterator_t<Base>>>;

    constexpr iterator& operator--();
    constexpr void operator--(int);
    constexpr iterator operator--(int)
      requires ref_is_glvalue && BidirectionalRange<Base> &&
        BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

    } // struct join_view<V>::iterator
  } // template class join_view<V>::iterator
} // namespace std::ranges

§ 24.7.7.3
friend constexpr bool operator==(const iterator& x, const iterator& y)  
requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&  
EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;

friend constexpr bool operator!=(const iterator& x, const iterator& y)  
requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&  
EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>>;

friend constexpr decltype(auto) iter_move(const iterator& i)  
noexcept(noexcept(ranges::iter_move(i.inner_))) {  
  return ranges::iter_move(i.inner_);  
}

friend constexpr void iter_swap(const iterator& x, const iterator& y)  
noexcept(noexcept(ranges::iter_swap(x.inner_, y.inner_)));

2 iterator::iterator_concept is defined as follows:

2.1 — If ref_is_glvalue is true,
2.1.1 — If Base and iter_reference_t<iterator_t<Base>> each model BidirectionalRange, then  
iterator_concept denotes bidirectional_iterator_tag.
2.1.2 — Otherwise, if Base and iter_reference_t<iterator_t<Base>> each model ForwardRange, then  
iterator_concept denotes forward_iterator_tag.
2.2 — Otherwise, iterator_concept denotes input_iterator_tag.

3 iterator::iterator_category is defined as follows:

3.1 — Let OUREMENT denote iterator_traits<iterator_t<Base>>::iterator_category, and let INNERC  
denote iterator_traits<iterator_t<iter_reference_t<iterator_t<Base>>>>::iterator_category.
3.2 — If ref_is_glvalue is true,
3.2.1 — If OUREMENT and INNERC each model DerivedFrom<bidirectional_iterator_tag>, iterator-  
category denotes bidirectional_iterator_tag.
3.2.2 — Otherwise, if OUREMENT and INNERC each model DerivedFrom<forward_iterator_tag>, iterator-  
category denotes forward_iterator_tag.
3.3 — Otherwise, iterator_category denotes input_iterator_tag.

4 iterator::difference_type denotes the type:

common_type_t<  
iter_difference_t<iterator_t<Base>>,  
iter_difference_t<iterator_t<iter_reference_t<iterator_t<Base>>>>>

5 join_view iterators use the satisfy function to skip over empty inner ranges.

constexpr void satisfy();  // exposition only

6 Effects: Equivalent to:

auto update_inner = [this](iter_reference_t<iterator_t<Base>> x) -> decltype(auto) {  
  if constexpr (ref_is_glvalue)  // x is a reference  
    return (x);  // (x) is an lvalue  
  else  
    return (parent_->inner_ = view::all(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<iter_reference_t<iterator_t<Base>>>();

constexpr iterator(Parent& parent, iterator_t<V> outer)

Effects: Initializes outer_ with outer and parent_ with &parent; then calls satisfy().

constexpr iterator(iterator<!Const> i)
requires Const && ConvertibleTo<iterator_t<V>, iterator_t<Base>> && ConvertibleTo<iterator_t<InnerRng>,
iterator_t<iter_reference_t<iterator_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>>;

Effects: Equivalent to return inner_;

constexpr iterator& operator++();

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 && ForwardRange<Base> && ForwardRange<iter_reference_t<iterator_t<Base>>>;

Effects: Equivalent to:
auto tmp = *this;
++*this;
return tmp;

constexpr iterator& operator--();
requires ref_is_glvalue && BidirectionalRange<Base> && BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

Effects: Equivalent to:
if (outer_ == ranges::end(*--outer_))
  inner_ = ranges::end(*--outer_);
while (inner_ == ranges::begin(*outer_))
  inner_ = ranges::end(*--outer_);
--inner_; return *this;

constexpr iterator operator--(int)
requires ref_is_glvalue && BidirectionalRange<Base> && BidirectionalRange<iter_reference_t<iterator_t<Base>>>;

Effects: Equivalent to:
auto tmp = *this;
--*this;
return tmp;
friend constexpr bool operator==(const iterator& x, const iterator& y)  
requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&  
EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>;  

Effects: Equivalent to: return x.outer_ == y.outer_ && x.inner_ == y.inner_;  

friend constexpr bool operator!=(const iterator& x, const iterator& y)  
requires ref_is_glvalue && EqualityComparable<iterator_t<Base>> &&  
EqualityComparable<iterator_t<iter_reference_t<iterator_t<Base>>>;  

Effects: Equivalent to: return !(x == y);  

friend constexpr void iter_swap(const iterator& x, const iterator& y)  
noexcept(noexcept(ranges::iter_swap(x.inner_, y.inner_));  

Effects: Equivalent to: return ranges::iter_swap(x.inner_, y.inner_);  

24.7.7.4 Class template join_view::sentinel  

namespace std::ranges {  
template<class 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 && ConvertibleTo<sentinel_t<V>, sentinel_t<Base>>;  
    friend constexpr bool operator==(const iterator<Const>& x, const sentinel& y);  
    friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);  
    friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);  
    friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);  
};  
}  

constexpr explicit sentinel(Parent& parent);  

Effects: Initializes end_ with ranges::end(parent.base_).  

constexpr sentinel(sentinel<!Const> s)  
requires Const && ConvertibleTo<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_;  

friend constexpr bool operator==(const sentinel& x, const iterator<Const>& y);  

Effects: Equivalent to: return y == x;  

friend constexpr bool operator!=(const iterator<Const>& x, const sentinel& y);  

Effects: Equivalent to: return !(x == y);  

friend constexpr bool operator!=(const sentinel& x, const iterator<Const>& y);  

Effects: Equivalent to: return !(y == x);  

24.7.7.5 view::join  
The name view::join denotes a range adaptor object (24.7.1). For some subexpression E, the expression  
view::join(E) is expression-equivalent to join_view{E}.  

§ 24.7.7.5 | 976
24.7.8 Split view

24.7.8.1 Overview

`split_view` takes a View and a delimiter, and splits the View into subranges on the delimiter. The delimiter can be a single element or a View of elements.

[Example:

```cpp
string str{"the quick brown fox"};
split_view sentence{str, ' '};
for (auto word : sentence) {
  for (char ch : word)
    cout << ch;
  cout << '*';
}
// The above prints: the*quick*brown*fox*
```
—end example]

24.7.8.2 Class template split_view

```cpp
namespace std::ranges {
  template<auto> struct require-constant; // exposition only

template<class R>
concept tiny-range = // exposition only
  SizedRange<R> &&
  requires { typename require-constant<remove_reference_t<R>::size>(); } &&
  (remove_reference_t<R>::size() <= 1);

template<InputRange V, ForwardRange Pattern>
requires View<V> && View<Pattern> &&
  IndirectlyComparable<iterator_t<V>, iterator_t<Pattern>, ranges::equal_to> &&
  (ForwardRange<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 !ForwardRange<V>
  // 24.7.8.3, class template split_view::outer_iterator
  template<bool> struct outer_iterator; // exposition only
  // 24.7.8.5, class template split_view::inner_iterator
  template<bool> struct inner_iterator; // exposition only
public:
  split_view() = default;
  constexpr split_view(V base, Pattern pattern); // exposition only

template<InputRange R, ForwardRange P>
requires Constructible<V, all_view<R>> &&
  Constructible<Pattern, all_view<P>>
constexpr split_view(R&& r, P&& p); // exposition only

template<InputRange R>
requires Constructible<V, all_view<R>> &&
  Constructible<Pattern, single_view<iter_value_t<iterator_t<R>>>>
constexpr split_view(R&& r, iter_value_t<iterator_t<R>> e); // exposition only

  constexpr auto begin() { // exposition only
    if constexpr (ForwardRange<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 ForwardRange<V> && ForwardRange<const V> {
    return outer_iterator<true>{*this, ranges::begin(base_)};
}

cconstexpr auto end() requires ForwardRange<V> && CommonRange<V> {
    return outer_iterator<simple-view<V>>{*this, ranges::end(base_)};
}

cconstexpr auto end() const {
    if constexpr (ForwardRange<V> && ForwardRange<const V> && CommonRange<const V>)
        return outer_iterator<true>{*this, ranges::end(base_)};
    else
        return default_sentinel;
};

template<class R, class P>
split_view(R&&, P&&) -> split_view<all_view<R>, all_view<P>>;

template<InputRange R>
split_view(R&&, iter_value_t<iterator_t<R>>)
    -> split_view<all_view<R>, single_view<iter_value_t<iterator_t<R>>>>;

constexpr split_view(V base, Pattern pattern);

Effects: Initializes base_ with std::move(base), and pattern_ with std::move(pattern).

template<InputRange R, ForwardRange P>
requires Constructible<V, all_view<R>> &&
Constructible<Pattern, all_view<P>>
constexpr split_view(R&& r, P&& p);

Effects: Initializes base_ with view::all(std::forward<R>(r)) and pattern_ with view::all(std::forward<P>(p)).

template<InputRange R>
requires Constructible<V, all_view<R>> &&
Constructible<Pattern, single_view<iter_value_t<iterator_t<R>>>>
constexpr split_view(R&& r, iter_value_t<iterator_t<R>> e);

Effects: Initializes base_ with view::all(std::forward<R>(r)) and pattern_ with single_view{std::move(e)}.

24.7.8.3 Class template split_view::outer_iterator [range.split.outer]

namespace std::ranges {
    template<class V, class 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 ForwardRange
                iterator_t<Base>();

        public:
            using iterator_concept = // exposition only
                conditional_t<ForwardRange<Base>, forward_iterator_tag, input_iterator_tag>;
            using iterator_category = input_iterator_tag; // see 24.7.8.4
            using value_type; // see 24.7.8.4
            using difference_type = iter_difference_t<iterator_t<Base>>;

§ 24.7.8.3 978
outer_iterator() = default;
constexpr explicit outer_iterator(Parent& parent)
    requires (!ForwardRange<Base>);
constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
    requires ForwardRange<Base>;
constexpr outer_iterator(outer_iterator<!Const> i)
    requires Const &&ConvertibleTo<iterator_t<V>, iterator_t<const V>>;

constexpr value_type operator*() const;
constexpr outer_iterator& operator++();
constexpr decltype(auto) operator++(int) {
    if constexpr (ForwardRange<Base>) {
        auto tmp = *this;
        +++this;
        return tmp;
    } else
        +++this;
}

friend constexpr bool operator==(const outer_iterator& x, const outer_iterator& y)
    requires ForwardRange<Base>;
friend constexpr bool operator!=(const outer_iterator& x, const outer_iterator& y)
    requires ForwardRange<Base>;
friend constexpr bool operator==(const outer_iterator& x, default_sentinel_t);
friend constexpr bool operator==(default_sentinel_t, const outer_iterator& x);
friend constexpr bool operator!=(const outer_iterator& x, default_sentinel_t y);
friend constexpr bool operator!=(default_sentinel_t y, const outer_iterator& x);
}

Many of the following specifications refer to the notional member current of outer_iterator. current is equivalent to current_ if V models ForwardRange, and parent_->current_ otherwise.

constexpr explicit outer_iterator(Parent& parent)
    requires (!ForwardRange<Base>);

Effects: Initializes parent_ with addressof(parent).

constexpr outer_iterator(Parent& parent, iterator_t<Base> current)
    requires ForwardRange<Base>;

Effects: Initializes parent_ with addressof(parent) and current_ with current.

constexpr outer_iterator(outer_iterator<!Const> i)
    requires Const &&ConvertibleTo<iterator_t<V>, iterator_t<const V>>;

Effects: Initializes parent_ with i.parent_ and current_ with std::move(i.current_).

constexpr value_type operator*() const;

Effects: Equivalent to: return value_type{*this};

constexpr outer_iterator& operator++();

Effects: Equivalent to:
    const auto end = ranges::end(parent_->base_);
    if (current == end) return *this;
    const auto [pbegin, pend] = subrange{parent_->pattern_};
    if (pbegin == pend) ++current;
    else {
        do {
            const auto [b, p] = ranges::mismatch(current, end, pbegin, pend);
            if (p == pend) {
                current = b; // The pattern matched; skip it
                break;
            }
        } while (p != pend);
    }

§ 24.7.8.3
while (++current != end);
return *this;

friend constexpr bool operator==(const outer_iterator& x, const outer_iterator& y)
requires ForwardRange<Base>;
7  Effects: Equivalent to: return x.current_ == y.current_;

friend constexpr bool operator!=(const outer_iterator& x, const outer_iterator& y)
requires ForwardRange<Base>;
8  Effects: Equivalent to: return !(x == y);

friend constexpr bool operator==(const outer_iterator& x, default_sentinel_t);
friend constexpr bool operator==(default_sentinel_t, const outer_iterator& x);
9  Effects: Equivalent to: return x.current_ == ranges::end(x.parent_->base_);

friend constexpr bool operator!=(const outer_iterator& x, default_sentinel_t y);
friend constexpr bool operator!=(default_sentinel_t y, const outer_iterator& x);
10 Effects: Equivalent to: return !(x == y);

24.7.8.4 Class split_view::outer_iterator::value_type

namespace std::ranges {
  template<class V, class Pattern>
  template<bool Const>
  struct split_view<V, Pattern>::outer_iterator<Const>::value_type {
    private:
      outer_iterator i_ = outer_iterator();  // exposition only
    public:
      value_type() = default;
      constexpr explicit value_type(outer_iterator i);
      constexpr inner_iterator<Const> begin() const;
      constexpr default_sentinel_t end() const;
    };

  constexpr explicit value_type(outer_iterator i);
1  Effects: Initializes i_ with i.

  constexpr inner_iterator<Const> begin() const;
2  Effects: Equivalent to: return inner_iterator<Const>{i_};

  constexpr default_sentinel_t end() const;
3  Effects: Equivalent to: return default_sentinel;

24.7.8.5 Class template split_view::inner_iterator

namespace std::ranges {
  template<class V, class 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 = iter_value_t<iterator_t<Base>>;
      using difference_type = iter_difference_t<iterator_t<Base>>;

      using iterator_concept = typename outer_iterator<Const>::iterator_concept;
      using iterator_category = see below;
      using value_type = iter_value_t<iterator_t<Base>>;
      using difference_type = iter_difference_t<iterator_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 (ForwardRange<V>) {
        auto tmp = *this;
        +++this;
        return tmp;
    } else
        +++this;
}

friend constexpr bool operator==(const inner_iterator& x, const inner_iterator& y)
    requires ForwardRange<Base>;
friend constexpr bool operator!=(const inner_iterator& x, const inner_iterator& y)
    requires ForwardRange<Base>;

friend constexpr bool operator==(const inner_iterator& x, default_sentinel_t);
friend constexpr bool operator==(default_sentinel_t, const inner_iterator& x);
friend constexpr bool operator!=(const inner_iterator& x, default_sentinel_t y);
friend constexpr bool operator!=(default_sentinel_t y, const inner_iterator& x);

friend constexpr decltype(auto) iter_move(const inner_iterator& i)
    noexcept(noexcept(ranges::iter_move(i.i_.current))) {
    return ranges::iter_move(i.i_.current);
}

friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
    noexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
    requires IndirectlySwappable<iterator_t<Base>>;
};

1 The typedef-name iterator_category denotes forward_iterator_tag if iterator_traits<iterator_t<Base>>::iterator_category models DerivedFrom<forward_iterator_tag>, and input_iterator_tag otherwise.

constexpr explicit inner_iterator(outer_iterator<Const> i);

2 Effects: Initializes i_ with i.

constexpr inner_iterator& operator++() const;

3 Effects: Equivalent to:
    incremented_ = true;
    if constexpr (!ForwardRange<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 ForwardRange<Base>;

4 Effects: Equivalent to: return x.i_.current_ == y.i_.current_;

friend constexpr bool operator!=(const inner_iterator& x, const inner_iterator& y)
    requires ForwardRange<Base>;

5 Effects: Equivalent to: return !(x == y);

§ 24.7.8.5
friend constexpr bool operator==(const inner_iterator& x, default_sentinel_t);
friend constexpr bool operator==(default_sentinel_t, const inner_iterator& x);

Effects: Equivalent to:
auto cur = x.i_.current;
auto end = ranges::end(x.i_.parent_->base_);
if (cur == end) return true;
auto [pcur, pend] = subrange(x.i_.parent_->pattern_);
if (pcur == pend) return x.incremented_;
do {
  if (*cur != *pcur) return false;
  if (++pcur == pend) return true;
} while (++cur != end);
return false;

friend constexpr bool operator!=(const inner_iterator& x, default_sentinel_t y);
friend constexpr bool operator!=(default_sentinel_t y, const inner_iterator& x);

Effects: Equivalent to: return !(x == y);

friend constexpr void iter_swap(const inner_iterator& x, const inner_iterator& y)
oexcept(noexcept(ranges::iter_swap(x.i_.current, y.i_.current)))
requires IndirectlySwappable<iterator_t<Base>>;

Effects: Equivalent to ranges::iter_swap(x.i_.current, y.i_.current).

24.7.8.6 view::split [range.split.adaptor]
The name view::split denotes a range adaptor object (24.7.1). For some subexpressions E and F, the expression view::split(E, F) is expression-equivalent to split_view(E, F).

24.7.9 Counted view [range.counted]
The name view::counted denotes a customization point object (16.4.2.2.6). Let E and F be expressions, and let T be decay_t< decltype((E)) >. Then the expression view::counted(E, F) is expression-equivalent to:
(1.1) IF T models Iterator and decltype((F)) models ConvertibleTo<iter_difference_t<T>>,
(1.1.1) subrange(E, E + static_cast<iter_difference_t<T>>(F)) if T models RandomAccessIterator.
(1.1.2) Otherwise, subrange{counted_iterator{E, F}, default_sentinel}.
(1.2) Otherwise, view::counted(E, F) is ill-formed. [Note: This case can result in substitution failure when view::counted(E, F) appears in the immediate context of a template instantiation. — end note]

24.7.10 Common view [range.common]
24.7.10.1 Overview [range.common.overview]
common_view takes a View which has different types for its iterator and sentinel and turns it into a View of the same elements with an iterator and sentinel of the same type.
[Note: common_view is useful for calling legacy algorithms that expect a range’s iterator and sentinel types to be the same. — end note]

[Example:
// Legacy algorithm:
template<class ForwardIterator>
size_t count(ForwardIterator first, ForwardIterator last);

template<ForwardRange R>
void my_algo(R&& r) {
  auto&& common = common_view(r);
  auto cnt = count(common.begin(), common.end());
  // ...
}
—end example]
24.7.10.2 Class template common_view

namespace std::ranges {
    template<View V>
    requires (!CommonRange<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<ViewableRange R>
            requires (!CommonRange<R> && Constructible<V, all_view<R>>)
            constexpr explicit common_view(R&& r);

            constexpr V base() const;

            constexpr auto size() requires SizedRange<V> { return ranges::size(base_); }

            constexpr auto size() const requires SizedRange<const V> { return ranges::size(base_); }

            constexpr auto begin() {
                if constexpr (RandomAccessRange<V> && SizedRange<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 (RandomAccessRange<const V> && SizedRange<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 (RandomAccessRange<V> && SizedRange<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 (RandomAccessRange<const V> && SizedRange<const V>)
                    return ranges::begin(base_) + ranges::size(base_);
                else
                    return common_iterator<iterator_t<const V>, sentinel_t<const V>>(ranges::end(base_));
            }
    }

    template<class R>
    common_view(R&&) -> common_view<all_view<R>>;
}

constexpr explicit common_view(V base);

Effects: Initializes base_ with std::move(base).

template<ViewableRange R>
requires (!CommonRange<R> && Constructible<V, all_view<R>>)

constexpr explicit common_view(R&& r);

Effects: Initializes base_ with view::all(std::forward<R>(r)).

constexpr V base() const;

Effects: Equivalent to: return base_;
template<class R>
    reverse_view(R&&) -> reverse_view<all_view<R>>;

constexpr explicit reverse_view(V base);

    Effects: Initializes base_ with std::move(base).

template<ViewableRange R>
    requires BidirectionalRange<R> && Constructible<V, all_view<R>>
constexpr explicit reverse_view(R&& r);

    Effects: Initializes base_ with view::all(std::forward<R>(r)).

constexpr V base() const;

    Effects: Equivalent to: return base_;

constexpr reverse_iterator<iterator_t<V>> begin();

    Returns:

    make_reverse_iterator(ranges::next(ranges::begin(base_), ranges::end(base_)))

    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 CommonRange<V>;
constexpr reverse_iterator<iterator_t<const V>> begin() const
    requires CommonRange<const V>;

    Effects: Equivalent to: return make_reverse_iterator(ranges::end(base_));

constexpr reverse_iterator<iterator_t<V>> end();
constexpr reverse_iterator<iterator_t<const V>> end() const
    requires CommonRange<const V>;

    Effects: Equivalent to: return make_reverse_iterator(ranges::begin(base_));

24.7.11.3 view::reverse [range.reverse.adaptor]

The name view::reverse denotes a range adaptor object (24.7.1). For some subexpression E, the expression view::reverse(E) is expression-equivalent to:

(1.1) — If the type of E is a (possibly cv-qualified) specialization of reverse_view, equivalent to E.base().
(1.2) — 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,

    (1.2.1) — if K is subrange_kind::sized, equivalent to:

    subrange<I, I, K>(E.end().base(), E.begin().base(), E.size())

    (1.2.2) — otherwise, equivalent to:

    subrange<I, I, K>(E.end().base(), E.begin().base())

    However, in either case E is evaluated only once.
(1.3) — Otherwise, equivalent to reverse_view(E).
25 Algorithms library

25.1 General

This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 22) and other sequences.

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 80.

Table 80 — Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.2</td>
<td></td>
</tr>
<tr>
<td>25.3</td>
<td></td>
</tr>
<tr>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td>25.6</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>25.7</td>
<td></td>
</tr>
<tr>
<td>25.9</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>25.10</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

25.2 Algorithms requirements

All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

The entities defined in the std::ranges namespace in this Clause are not found by argument-dependent name lookup (6.4.2). When found by unqualified (6.4.1) 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 (13.6.6.2) than std::ranges::find since the former requires its first two parameters to have the same type. — end example]

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.

Throughout this Clause, where the template parameters are not constrained, the names of template parameters are used to express type requirements.

1. If an algorithm’s template parameter is named InputIterator, InputIterator1, or InputIterator2, the template argument shall satisfy the Cpp17InputIterator requirements (23.3.5.2).
2. If an algorithm’s template parameter is named OutputIterator, OutputIterator1, or OutputIterator2, the template argument shall satisfy the Cpp17OutputIterator requirements (23.3.5.3).
3. If an algorithm’s template parameter is named ForwardIterator, ForwardIterator1, or ForwardIterator2, the template argument shall satisfy the Cpp17ForwardIterator requirements (23.3.5.4).
4. If an algorithm’s template parameter is named BidirectionalIterator, BidirectionalIterator1, or BidirectionalIterator2, the template argument shall satisfy the Cpp17BidirectionalIterator requirements (23.3.5.5).
— If an algorithm’s template parameter is named `RandomAccessIterator`, `RandomAccessIterator1`, or `RandomAccessIterator2`, the template argument shall satisfy the `Cpp17RandomAccessIterator` 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 argument shall satisfy 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 movable, 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.234 When such a version is provided for `algorithm` it is called `algorithm_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

7 When not otherwise constrained, the `Predicate` parameter is used whenever an algorithm expects a function object (20.14) that, when applied to the result of dereferencing the corresponding iterator, returns a value testable as `true`. In other words, if an algorithm takes `Predicate pred` as its argument and `first` as its iterator argument with value type `T`, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (7.3). The function object `pred` shall not apply any non-constant function through the dereferenced iterator. Given a glvalue `u` of type (possibly `const`) `T` that designates the same object as `*first`, `pred(u)` shall be a valid expression that is equal to `pred(*first)`.

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 other words, if an algorithm takes `BinaryPredicate binary_pred` as its argument and `first1` and `first2` as its iterator arguments with respective value types `T1` and `T2`, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (7.3). Unless otherwise specified, `BinaryPredicate` always takes the first iterator’s `value_type` as its first argument, that is, in those cases when `T` value is part of the signature, it should work correctly in the construct `binary_pred(*first1, value)` contextually converted to `bool` (7.3). `binary_pred` shall not apply any non-constant function through the dereferenced iterators. Given a glvalue `u` of type (possibly `const`) `T1` that designates the same object as `*first1`, and a glvalue `v` of type (possibly `const`) `T2` that designates the same object as `*first2`, `binary_pred(u, *first2)`, `binary_pred(*first1, v)`, and `binary_pred(u, v)` shall each be a valid expression that is equal to `binary_pred(*first1, *first2)`, and `binary_pred(u, value)` shall be a valid expression that is equal to `binary_pred(*first1, value)`.

9 The parameters `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, and `BinaryOperation2` are used whenever an algorithm expects a function object (20.14).

10 [Note: 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]

11 When the description of an algorithm gives an expression such as `*first == value` for a condition, the expression shall evaluate to either `true` or `false` in boolean contexts.

12 In the description of the algorithms, operator `+` is used for some of the iterator categories for which it does not have to be defined. In these cases the semantics of `a + n` are the same as those of

```cpp
auto tmp = a;
for (; n < 0; ++n) --tmp;
for (; n > 0; --n) ++tmp;
return tmp;
```

Similarly, operator `-` is used for some combinations of iterators and sentinel types for which it does not have to be defined. If `[a, b)` denotes a range, the semantics of `b - a` in these cases are the same as those of

```cpp
iter_difference_t<remove_reference_t<decltype(a)>> n = 0;
for (auto tmp = a; tmp != b; ++tmp) ++n;
return n;
```

234) 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`.
and if \([b, a)\) denotes a range, the same as those of

\[
\text{iter\_difference_t<remove\_reference_t<\text{decltype(b)>> n = 0;
for (auto tmp = b; tmp != a; ++tmp) --n;
return n;
}\]

In the description of algorithm return values, a sentinel value \(s\) denoting the end of a range \([i, s)\) is sometimes returned when 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 \text{Range} arguments (24.4.2) behave as if they are implemented by calling \(\text{ranges::begin and ranges::end on the Range(s) and dispatching to the overload in namespace ranges}\) 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]

The class templates \text{binary\_transform\_result}, \text{for\_each\_result}, \text{minmax\_result}, \text{mismatch\_result}, \text{copy\_result}, and \text{partition\_copy\_result} have the template parameters, data members, and special members specified above. They have no base classes or members other than those specified.

25.3 Parallel algorithms [algorithms.parallel]

This subclause describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

25.3.1 Terms and definitions [algorithms.parallel.defns]

1 A parallel algorithm is a function template listed in this document with a template parameter named \text{ExecutionPolicy}.

2 Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

(2.1) — All operations of the categories of the iterators that the algorithm is instantiated with.

(2.2) — Operations on those sequence elements that are required by its specification.

(2.3) — User-provided function objects to be applied during the execution of the algorithm, if required by the specification.

(2.4) — 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:

(2.5) — 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}.

(2.6) — The \text{swap} function on the elements of the sequence (as per the preconditions specified in 25.7.1.1).

(2.7) — The user-provided \text{Compare} function object.

—end example]

3 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:

\[
\begin{align*}
\text{int x = 0;}
\text{std::mutex m;}
\text{void f() \{}\\
\text{int a[] = \{1,2\};}
\text{std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [\&](int) \{}\\
\text{std::lock\_guard<mutex> guard(m); // incorrect: lock\_guard constructor calls m.lock()}
\text{++x;}
\text{\})};
\end{align*}
\]

§ 25.3.1 988
The above program may result in two consecutive calls to \texttt{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 [algorithms.parallel.user]  
1 Unless otherwise specified, function objects passed into parallel algorithms as objects of type \texttt{Predicate}, \texttt{BinaryPredicate}, \texttt{Compare}, \texttt{UnaryOperation}, \texttt{BinaryOperation1}, \texttt{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 [algorithms.parallel.exec]  
1 Parallel algorithms have template parameters named \texttt{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.

2 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, \texttt{swap()}, ++, --, \&=, and assignments modify the object. For the assignment and \&= operators, only the left argument is modified. — end note]

3 Unless otherwise stated, implementations may make arbitrary copies of elements (with type \texttt{T}) from sequences where \texttt{is_trivially_copy_constructible_v<T>} and \texttt{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 \texttt{reference_wrapper<T>} (20.14.5) or some equivalent solution. — end note]

4 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type \texttt{execution::sequenced_policy} all occur in the calling thread of execution. [Note: The invocations are not interleaved; see 6.8.1. — end note]

5 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type \texttt{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 6.8.1 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 \texttt{execution::unsequenced_policy} algorithm. [Note: Because \texttt{execution::unsequenced_policy} allows the execution of element access functions to be interleaved on a single thread of execution, blocking synchronization, including the use of mutexes, risks deadlock. — end note]

6 The invocations of element access functions in parallel algorithms invoked with an execution policy object of type \texttt{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 \texttt{thread} (32.3.2) provide concurrent forward progress guarantees (6.8.2.2), 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:  
\begin{verbatim}  
int a[] = {0,1};  
std::vector<int> v;  
std::for_each(std::execution::par, std::begin(a), std::end(a), \[&\](int i) {  
v.push_back(i*2+1);  
// incorrect: data race  
});  
\end{verbatim}  
The program above has a data race because of the unsynchronized access to the container \texttt{v}. — end example]

[Example:  
\begin{verbatim}  
std::atomic<int> x{0};  
int a[] = {1,2};  
\end{verbatim}  
]
The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `std::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 6.8.1 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 `std::execution::parallel_unsequenced_policy` algorithm. 

[Note: Because `std::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]

The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

25.3.4 Parallel algorithm exceptions 

During the execution of a parallel algorithm, if temporary memory resources are required for parallelization and none are available, the algorithm throws a `std::system_error` exception.

During the execution of a parallel algorithm, if the invocation of an element access function exits via an uncaught exception, the behavior is determined by the `std::execution::parallel::algorithm` overload.

25.3.5 ExecutionPolicy algorithm overloads

Parallel algorithms are algorithm overloads. Each parallel algorithm overload has an additional template type parameter named `ExecutionPolicy`, which is the first template parameter. Additionally, each parallel algorithm overload has an additional function parameter of type `std::execution::parallel::algorithm`, which is the first function parameter. [Note: Not all algorithms have parallel algorithm overloads. —end note]

Unless otherwise specified, the semantics of `std::execution::parallel::algorithm` overload 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 \( O(\text{expr}) \).

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 {
  // 25.5, non-modifying sequence operations
  // 25.5.1, 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<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool all_of(I first, S last, Pred pred, Proj proj = {});
    template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool all_of(R&& r, Pred pred, Proj proj = {});
  }

  // 25.5.2, 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<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool any_of(I first, S last, Pred pred, Proj proj = {});
    template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool any_of(R&& r, Pred pred, Proj proj = {});
  }

  // 25.5.3, 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, // see 25.3.5
    ForwardIterator first, ForwardIterator last, Predicate pred);

  namespace ranges {
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool none_of(I first, S last, Pred pred, Proj proj = {});
    template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr bool none_of(R&& r, Pred pred, Proj proj = {});
  }

  // 25.5.4, for each
  template<class InputIterator, class Function>
  constexpr Function for_each(InputIterator first, InputIterator last, Function f);

§ 25.4
template<class ExecutionPolicy, class ForwardIterator, class Function>
void for_each(ExecutionPolicy&& exec,  // see 25.3.5
    ForwardIterator first, ForwardIterator last, Function f);

namespace ranges {
    template<class I, class F>
    struct for_each_result {
        [[no_unique_address]] I in;
        [[no_unique_address]] F fun;

        template<class I2, class F2>
        requires ConvertibleTo<const I&, I2> && ConvertibleTo<const F&, F2>
        operator for_each_result<I2, F2>() const & {
            return {in, fun};
        }

        template<class I2, class F2>
        requires ConvertibleTo<I, I2> && ConvertibleTo<F, F2>
        operator for_each_result<I2, F2>() && {
            return {std::move(in), std::move(fun)};
        }
    };

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectUnaryInvocable<projected<I, Proj>> Fun>
    constexpr for_each_result<I, Fun>
    for_each(I first, S last, Fun f, Proj proj = {});

    template<InputRange R, class Proj = identity,
        IndirectUnaryInvocable<projected<iterator_t<R>, Proj>> Fun>
    constexpr for_each_result<safe_iterator_t<R>, Fun>
    for_each(R&& r, Fun f, Proj proj = {});
}

template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);

template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
ForwardIterator for_each_n(ExecutionPolicy&& exec,  // see 25.3.5
    ForwardIterator first, ForwardIterator last, Function f);

// 25.5.5, 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 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,  // see 25.3.5
    ForwardIterator first, ForwardIterator last, Predicate pred);
namespace ranges {
    template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>constexpr I find(I first, S last, const T& value, Proj proj = {});
    template<InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>constexpr safe_iterator_t<R> find(R&& r, const T& value, Proj proj = {});
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if(I first, S last, Pred pred, Proj proj = {});
    template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R> find_if(R&& r, Pred pred, Proj proj = {});
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if_not(I first, S last, Pred pred, Proj proj = {});
    template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R> find_if_not(R&& r, Pred pred, Proj proj = {});
}

// 25.5.6, 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);

namespace ranges {
    template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
    class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<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<ForwardRange R1, ForwardRange R2,
    class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
    requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
    constexpr safe_subrange_t<R1>
    find_end(R1&& r1, R2&& r2, Pred pred = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});
}
// 25.5.7, find first
template<class InputIterator, class ForwardIterator>
constexpr InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2);

template<class InputIterator, class ForwardIterator, class BinaryPredicate>
constexpr InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2,
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_first_of(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_first_of(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2,
   class Proj1 = identity, class Proj2 = identity,
   IndirectRelation<projected<I1, Proj1>,
   projected<I2, Proj2>> Pred = ranges::equal_to>
  constexpr I1 find_first_of(I1 first1, S1 last1, I2 first2, S2 last2,
   Pred pred = {},
   Proj1 proj1 = {}, Proj2 proj2 = {});
  template<InputRange R1, ForwardRange R2, class Proj1 = identity, class Proj2 = identity,
   IndirectRelation<iterator_t<R1>, Proj1>,
   projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to>
  constexpr safe_iterator_t<R1>
  find_first_of(R1&& r1, R2&& r2,
   Pred pred = {},
   Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.5.8, 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<ForwardIterator I, Sentinel<I> S, class Proj = identity,
   IndirectRelation<projected<I, Proj>> Pred = ranges::equal_to>
  constexpr I adjacent_find(I first, S last, Pred pred = {},
   Proj proj = {});
}

§ 25.4 994
template<ForwardRange R, class Proj = identity,
    IndirectRelation<ranges::equal_to, projected<range_iterator_t<R>>, Proj>> Pred = ranges::equal_to>
constexpr safe_iterator_t<R>
adjacent_find(R&& r, Pred pred = {}, Proj proj = {});
}

// 25.5.9, 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>
constexpr typename iterator_traits<ForwardIterator>::difference_type
count(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator first, ForwardIterator last, const T& value);

namespace ranges {
    template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<I>, Proj>> const
    constexpr iter_difference_t<I>
count(I first, S last, const T& value, Proj proj = {});

    template<InputRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<range_iterator_t<R>>, Proj>> const
    constexpr iter_difference_t<range_iterator_t<R>>
count(R&& r, const T& value, Proj proj = {});

    template<InputIterator I, Sentinel<I> S, class Proj = identity,
        IndirectUnaryPredicate<projected<I>, Proj>> Pred>
    constexpr iter_difference_t<I>
count_if(I first, S last, Pred pred, Proj proj = {});

    template<InputRange R, class Proj = identity,
        IndirectUnaryPredicate<projected<range_iterator_t<R>>, Proj>> Pred>
    constexpr iter_difference_t<range_iterator_t<R>>
count_if(R&& r, Pred pred, Proj proj = {});
}

// 25.5.10, mismatch
template<class InputIterator1, class InputIterator2>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2);

template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec, // see 25.3.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

namespace ranges {
  template<class I1, class I2>
  struct mismatch_result {
    [[no_unique_address]] I1 in1;
    [[no_unique_address]] I2 in2;

    template<class II1, class II2>
    requires ConvertibleTo<const I1&, II1> && ConvertibleTo<const I2&, II2>
    operator mismatch_result<II1, II2>() const & {
      return {in1, in2};
    }

    template<class II1, class II2>
    requires ConvertibleTo<I1, II1> && ConvertibleTo<I2, II2>
    operator mismatch_result<II1, II2>() && {
      return {std::move(in1), std::move(in2)};
    }
  };

  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<I1, Proj1>, projected<I2, Proj2>> Pred = ranges::equal_to>
  constexpr mismatch_result<I1, I2>
  mismatch(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {});

  template<InputRange R1, InputRange R2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectRelation<projected<iterator_t<R1>, Proj1>, projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to>
  constexpr mismatch_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
  mismatch(R1&& r1, R2&& r2, Pred pred = {});
}

// 25.5.11, 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);

§ 25.4
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
bool equal(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
bool equal(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
bool equal(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
bool equal(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

namespace ranges {
    template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
constexpr bool equal(I1 first1, S1 last1, I2 first2, S2 last2,
Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});

template<InputRange R1, InputRange R2, class Pred = ranges::equal_to,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
constexpr bool equal(R1&& r1, R2&& r2, Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.5.12, is permutation
namespace ranges {
    template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
Sentinel<I2> S2, class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
constexpr bool is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});

template<ForwardRange R1, ForwardRange R2, class Pred = ranges::equal_to,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
constexpr bool is_permutation(R1&& r1, R2&& r2, Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});
}

§ 25.4
constexpr bool is_permutation(R1&& r1, R2&& r2, Pred pred = {},
    Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.5.13, 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);

namespace ranges {
  template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
    Sentinel<I2> S2, class Pred = ranges::equal_to,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
  constexpr subrange<I1>
  search(I1 first1, S1 last1, I2 first2, S2 last2, Pred pred = {},
    Proj1 proj1 = {}, Proj2 proj2 = {});

  template<ForwardRange R1, ForwardRange R2, class Pred = ranges::equal_to,
    class Proj1 = identity, class Proj2 = identity>
  requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
  constexpr safe_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);

§ 25.4
namespace ranges {

template<ForwardIterator I, Sentinel<I> S, class T,
    class Pred = ranges::equal_to, class Proj = identity>
    requires IndirectlyComparable<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<ForwardRange R, class T, class Pred = ranges::equal_to,
    class Proj = identity>
    requires IndirectlyComparable<iterator_t<R>, const T*, Pred, Proj>
    constexpr safe_subrange_t<R>
    search_n(R&& r, iter_difference_t<iterator_t<R>> count,
        const T& value, Pred pred = {}, Proj proj = {});
}

namespace ranges {

template<class ForwardIterator, class Searcher>
    constexpr ForwardIterator
    search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);
}

// 25.6, mutating sequence operations
// 25.6.1, copy

namespace ranges {

template<class InputIterator, class OutputIterator>
    constexpr OutputIterator copy(InputIterator first, InputIterator last,
        OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
    ForwardIterator2 copy(ExecutionPolicy&& exec,
        ForwardIterator1 first, ForwardIterator1 last,
        ForwardIterator2 result);
}

namespace ranges {

template<class I, class O>
    struct copy_result {
        [[no_unique_address]] I in;
        [[no_unique_address]] O out;

        template<class I2, class O2>
            requires ConvertibleTo<const I&, I2> && ConvertibleTo<const O&, O2>
            operator copy_result<I2, O2>() const & {
                return {in, out};
            }

        template<class I2, class O2>
            requires ConvertibleTo<I, I2> && ConvertibleTo<O, O2>
            operator copy_result<I2, O2>() && {
                return {std::move(in), std::move(out)};
            }
    };

    template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
        requires IndirectlyCopyable<I, O>
        constexpr copy_result<I, O>
        copy(I first, S last, O result);

    template<InputRange R, WeaklyIncrementable O>
        requires IndirectlyCopyable<iterator_t<R>, O>
        constexpr copy_result<safe_iterator_t<R>, O>
        copy(R&& r, O result);
}

namespace ranges {

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, ForwardIterator1 last,
        ForwardIterator2 result);
}
namespace ranges {
    template<class I, class O>
    using copy_n_result = copy_result<I, O>;

    template<InputIterator I, WeaklyIncrementable O>
    requires IndirectlyCopyable<I, O>
    constexpr copy_n_result<I, O> copy_n(I first, iter_difference_t<I> n, O result);
}

namespace ranges {
    template<class InputIterator, class OutputIterator, class Predicate>
    constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
        OutputIterator result, Predicate pred);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
        class Predicate>
    ForwardIterator2 copy_if(ExecutionPolicy&& exec, // see §25.3.5
        ForwardIterator1 first, ForwardIterator1 last,
        ForwardIterator2 result, Predicate pred);
}

namespace ranges {
    template<class I1, class I2>
    using copy_backward_result = copy_result<I1, I2>;

    template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyCopyable<I1, I2>
    constexpr copy_backward_result<I1, I2> copy_backward(I1 first, S1 last, I2 result);

    template<BidirectionalRange R, BidirectionalIterator I>
    requires IndirectlyCopyable<iterator_t<R>, I>
    constexpr copy_backward_result<safe_iterator_t<R>, I> copy_backward(R&& r, I result);
}

// §25.4

// 25.6.2, move
template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last,
    OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 move(ExecutionPolicy&& exec, // see §25.3.5
    ForwardIterator2 result);
namespace ranges {
    template<class I, class O>
    using move_result = copy_result<I, O>;

    template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
    requires IndirectlyMovable<I, O>
    constexpr move_result<I, O> move(I first, S last, O result);

    template<InputRange R, WeaklyIncrementable O>
    requires IndirectlyMovable<iterator_t<R>, O>
    constexpr move_result<safe_iterator_t<R>, O> move(R&& r, O result);
}

namespace ranges {
    template<class BidirectionalIterator1, class BidirectionalIterator2>
    constexpr BidirectionalIterator2 move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
        BidirectionalIterator2 result);

    template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
    requires IndirectlyMovable<I1, I2>
    constexpr move_backward_result<I1, I2> move_backward(I1 first, S1 last, I2 result);

    template<BidirectionalRange R, BidirectionalIterator I>
    requires IndirectlyMovable<iterator_t<R>, I>
    constexpr move_backward_result<safe_iterator_t<R>, I> move_backward(R&& r, I result);
}

// 25.6.3, swap
namespace ranges {
    template<class I1, class I2>
    using swap_ranges_result = mismatch_result<I1, I2>;

    template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2>
    requires IndirectlySwappable<I1, I2>
    constexpr swap_ranges_result<I1, I2> swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

    template<InputRange R1, InputRange R2>
    requires IndirectlySwappable<iterator_t<R1>, iterator_t<R2>>
    constexpr swap_ranges_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
        swap_ranges(R1&& r1, R2&& r2);
}

namespace ranges {
    template<class ForwardIterator1, class ForwardIterator2>
    constexpr void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
}
template<class InputIterator, class OutputIterator, class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first1, InputIterator last1,
OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2, class OutputIterator,
class BinaryOperation>
constexpr OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result,
BinaryOperation binary_op);

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, ForwardIterator2 last2, ForwardIterator result,
BinaryOperation binary_op);

namespace ranges {

template<class I, class O>
using unary_transform_result = copy_result<I, O>;//

// 25.6.4, transform

template<class InputIterator, class OutputIterator, class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first1, InputIterator last1,
OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2, class OutputIterator,
class BinaryOperation>
constexpr OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, OutputIterator result,
BinaryOperation binary_op);

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, ForwardIterator2 last2, ForwardIterator result,
BinaryOperation binary_op);

namespace ranges {

template<class I1, class I2, class O>
using binary_transform_result = copy_result<I1, I2, O>;

struct binary_transform_result {
  [no_unique_address] I1 in1;
  [no_unique_address] I2 in2;
  [no_unique_address] O out;

  template<class II1, class II2, class OO>
  requires ConvertibleTo<const I1&, II1> &&
  ConvertibleTo<const I2&, II2> && ConvertibleTo<const O&, OO>
operator binary_transform_result<II1, II2, OO>() const & {
    return {in1, in2, out};
  }

  template<class II1, class II2, class OO>
  requires ConvertibleTo<const I1&, II1> &&
  ConvertibleTo<const II2, I2> && ConvertibleTo<const O&, OO>
operator binary_transform_result<II1, II2, OO>() const && {
    return {std::move(in1), std::move(in2), std::move(out)};
  }
};

§ 25.4

1002

© ISO/IEC
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, 
WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity, 
class Proj2 = identity>
requires 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<InputRange R1, InputRange R2, WeaklyIncrementable O, 
CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_t<F&>, projected<iterator_t<R1>, Proj1>, 
projected<iterator_t<R2>, Proj2> >
constexpr binary_transform_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
transform(R1&& r1, R2&& r2, O result, 
F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});
}

// 25.6.5, 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<InputIterator I, Sentinel<I> S, class T1, class T2, class Proj = identity>
requires Writable<I, const T2&> &&
IndirectRelation<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<InputRange R, class T1, class T2, class Proj = identity>
requires Writable<iterator_t<R>, const T2&> &&
IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T1*> 
constexpr safe_iterator_t<R> replace(R&& r, const T1& old_value, const T2& new_value, Proj proj = {});
template<InputIterator I, Sentinel<I> S, class T, class Proj = identity, 
IndirectUnaryPredicate<projected<I, Proj>> Pred>
requires Writable<I, const T&>
constexpr I replace_if(I first, S last, Pred pred, const T& new_value, Proj proj = {});
template<InputRange R, class T, class Proj = identity, 
IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
requires Writable<iterator_t<R>, const T&>
constexpr safe_iterator_t<R> replace_if(R&& r, Pred pred, const T& new_value, Proj proj = {});
}

template<InputIterator I, Sentinel<I> S1, InputIterator I2, Sentinel<I2> S2, 
WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity, 
class Proj2 = identity>
requires Writable<O, indirect_result_t<F&>, projected<I1, Proj1>, 
projected<I2, Proj2> >
ForwardIterator1 first, ForwardIterator1 last, 
ForwardIterator2 result, 
const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last, OutputIterator result, Predicate pred, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Predicate, class T>
ForwardIterator2 replace_copy_if(ExecutionPolicy& exec, ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result, Predicate pred, const T& new_value);

namespace ranges {
    template<class I, class O>
    using replace_copy_result = copy_result<I, O>;

    template<InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O, class Proj = identity>
    requires IndirectlyCopyable<I, O> && IndirectRelation<ranges::equal_to, projected<I, Proj>, const T1*>
    constexpr replace_copy_result<I, O> replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value, Proj proj = {});

    template<InputRange R, class T1, class T2, OutputIterator<const T2&> O, class Proj = identity>
    requires IndirectlyCopyable<iterator_t<R>, O> && IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T1*>
    constexpr replace_copy_result<safe_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 = copy_result<I, O>;

    template<InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O, class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O>
    constexpr replace_copy_if_result<I, O> replace_copy_if(I first, S last, O result, Pred pred, const T& new_value, Proj proj = {});

    template<InputRange R, class T, OutputIterator<const T&> O, class Proj = identity, IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr replace_copy_if_result<safe_iterator_t<R>, O> replace_copy_if(R&& r, O result, Pred pred, const T& new_value, Proj proj = {});
}

// 25.6.6, 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, OutputIterator<const T&> O, Sentinel<O> S>
    constexpr O fill(O first, S last, const T& value);
}

§ 25.4 1004
template<class T, OutputRange<const T&> R>
constexpr safe_iterator_t<R> fill(R&& r, const T& value);

template<class T, OutputIterator<const T&> O>
constexpr O fill_n(O first, iter_difference_t<O> n, const T& value);

} // 25.6.7, generate

namespace ranges {
    template<Iterator O, Sentinel<O> S, CopyConstructible F>
    requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    constexpr O generate(O first, S last, F gen);

    template<class R, CopyConstructible F>
    requires Invocable<F&> && OutputRange<R, invoke_result_t<F&>>
    constexpr safe_iterator_t<R> generate(R&& r, F gen);

    template<Iterator O, CopyConstructible F>
    requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
    constexpr O generate_n(O first, iter_difference_t<O> n, F gen);
}

// 25.6.8, remove

namespace ranges {
    template<Permutable I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*> 
    constexpr I remove(I first, S last, const T& value, Proj proj = {});

    template<ForwardRange R, class T, class Proj = identity, 
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R> remove(R&& r, const T& value, Proj proj = {});

    template<Permutable I, Sentinel<I> S, class Proj = identity, 
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I remove_if(I first, S last, Pred pred, Proj proj = {});

    template<ForwardRange R, class Proj = identity, 
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R> remove_if(R&& r, const T& value, Proj proj = {});

} // see 25.3.5

§ 25.4 1005
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 = copy_result<I, O>;

using remove_copy_result = copy_result<I, O>;

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
class Proj = identity>
requires IndirectlyCopyable<I, O> &&
IndirectRelation<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<InputRange R, WeaklyIncrementable O, class T, class Proj = identity>
requires IndirectlyCopyable<iterator_t<R>, O> &&
IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
constexpr remove_copy_result<safe_iterator_t<R>, O>
remove_copy(R&& r, O result, const T& value, Proj proj = {});

namespace ranges {

using remove_copy_if_result = copy_result<I, O>;

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
requires IndirectlyCopyable<I, O>
constexpr remove_copy_if_result<I, O>
remove_copy_if(I first, S last, O result, Pred pred, Proj proj = {});

template<InputRange R, WeaklyIncrementable O, class Proj = identity,
IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
requires IndirectlyCopyable<iterator_t<R>, O>
constexpr remove_copy_if_result<safe_iterator_t<R>, O>
remove_copy_if(R&& r, O result, Pred pred, Proj proj = {});
}

// 25.6.9, unique

template<class ForwardIterator>
constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
constexpr ForwardIterator unique(ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator unique(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last);

§ 25.4 1006
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);

namespace ranges {
    template<Permutable I, Sentinel<I> S, class Proj = identity,
             IndirectRelation<projected<I, Proj>> C = ranges::equal_to>
    constexpr I unique(I first, S last, C comp = {}, Proj proj = {});
    template<ForwardRange R, class Proj = identity,
             IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>
    constexpr safe_iterator_t<R> unique(R&& r, C comp = {}, Proj proj = {});
}

template<class InputIterator, class OutputIterator>
constexpr OutputIterator unique_copy(InputIterator first, InputIterator last,
OutputIterator result);

namespace ranges {
    template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
             class Proj = identity, IndirectRelation<projected<I, Proj>> C = ranges::equal_to>
    requires IndirectlyCopyable<I, O> &&
    (ForwardIterator<I> ||
    (InputIterator<O> && Same<iter_value_t<I>, iter_value_t<O>>) ||
    IndirectlyCopyableStorable<I, O>)
    constexpr unique_copy_result<I, O> unique_copy(I first, S last, O result, C comp = {}, Proj proj = {});
    template<InputRange R, WeaklyIncrementable O, class Proj = identity,
             IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>
    requires IndirectlyCopyable<iterator_t<R>, O> &&
    (ForwardIterator<iterator_t<R>> ||
    (InputIterator<O> && Same<iter_value_t<iterator_t<R>>, iter_value_t<O>>) ||
    IndirectlyCopyableStorable<iterator_t<R>, O>)
    constexpr unique_copy_result<safe_iterator_t<R>, O> unique_copy(R&& r, O result, C comp = {}, Proj proj = {});
}

// 25.6.10, reverse
template<class BidirectionalIterator>
constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator>
void reverse(ExecutionPolicy&& exec, // see 25.3.5
BidirectionalIterator first, BidirectionalIterator last);
namespace ranges {
    template<BidirectionalIterator I, Sentinel<I> S>
        requires Permutable<I>
    constexpr I reverse(I first, S last);
    template<BidirectionalRange R>
        requires Permutable<iterator_t<R>>
    constexpr safe_iterator_t<R> reverse(R&& r);
}

template<class BidirectionalIterator, class OutputIterator>
    constexpr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
    OutputIterator result);

namespace ranges {
    template<class I, class O>
        using reverse_copy_result = copy_result<I, O>;
    template<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
        requires IndirectlyCopyable<I, O>
    constexpr reverse_copy_result<I, O>
        reverse_copy(I first, I middle, S last, O result);
    template<BidirectionalRange R, WeaklyIncrementable O>
        requires IndirectlyCopyable<iterator_t<R>, O>
    constexpr reverse_copy_result<safe_iterator_t<R>, O>
        reverse_copy(R&& r, O result);
}

// 25.6.11, rotate
template<class ForwardIterator>
    constexpr ForwardIterator rotate(ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);

namespace ranges {
    template<Permutable I, Sentinel<I> S>
        requires Permutable<I>
    constexpr subrange<I> rotate(I first, I middle, S last);
    template<ForwardRange R>
        requires Permutable<iterator_t<R>>
    constexpr safe_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, // see 25.3.5
    ForwardIterator1 first, ForwardIterator1 middle,
    ForwardIterator2 last, ForwardIterator2 result);
namespace ranges {
  template<class I, class O>
  using rotate_copy_result = copy_result<I, O>;

  template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>
  requires IndirectlyCopyable<I, O>
  constexpr rotate_copy_result<I, O>
  rotate_copy(I first, I middle, S last, O result);

  template<ForwardRange R, WeaklyIncrementable O>
  requires IndirectlyCopyable<iterator_t<R>, O>
  constexpr rotate_copy_result<safe_iterator_t<R>, O>
  rotate_copy(R&& r, iterator_t<R> middle, O result);
}

// 25.6.12, sample
template<class PopulationIterator, class SampleIterator, class Distance, class UniformRandomBitGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last, SampleIterator out, Distance n, UniformRandomBitGenerator&& g);

// 25.6.13, shuffle
template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last, UniformRandomBitGenerator&& g);

namespace ranges {
  template<RandomAccessIterator I, Sentinel<I> S, class Gen>
  requires Permutable<I> &&
  UniformRandomBitGenerator<remove_reference_t<Gen>> &&
  ConvertibleTo<invoke_result_t<Gen&&>, iter_difference_t<I>><I>
  I shuffle(I first, S last, Gen&& g);
  template<RandomAccessRange R, class Gen>
  requires Permutable<iterator_t<R>> &&
  UniformRandomBitGenerator<remove_reference_t<Gen>> &&
  ConvertibleTo<invoke_result_t<Gen&&>, iter_difference_t<iterator_t<R>>><R>
  safe_iterator_t<R> shuffle(R&& r, Gen&& g);
}

// 25.6.14, 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, // see 25.3.5
             ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);

template<class ForwardIterator>
constexpr ForwardIterator
  shift_right(ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
  shift_right(ExecutionPolicy&& exec, // see 25.3.5
              ForwardIterator first, ForwardIterator last, typename iterator_traits<ForwardIterator>::difference_type n);

// 25.7, sorting and related operations
// 25.7.1, 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, // see 25.3.5
RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(ExecutionPolicy&& exec, // see 25.3.5
RandomAccessIterator first, RandomAccessIterator last, Compare comp);

namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R> sort(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R> sort(R&& r, Comp comp = {}, Proj proj = {});
}

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<RandomAccessIterator I, Sentinel<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<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R> sort(R&& r, Comp comp = {}, Proj proj = {});
}

template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
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);
§ 25.4
RandomAccessIterator last, Compare comp);
# 25.7.2, Nth element

```cpp
template<class RandomAccessIterator>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                          RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                          RandomAccessIterator last, Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator>
void nth_element(ExecutionPolicy&& exec, // see 25.3.5
                RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void nth_element(ExecutionPolicy&& exec, // see 25.3.5
                RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last, Compare comp);
```

§ 25.4 1012
namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R>
    nth_element(R&& r, iterator_t<R> nth, Comp comp = {}, Proj proj = {});
}

// 25.7.3, binary search
template<class ForwardIterator, class T>
constexpr ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
            const T& value);

template<class ForwardIterator, class T, class Compare>
constexpr ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
            const T& value, Compare comp);

namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr I lower_bound(I first, S last, const T& value, Comp comp = {},
                             Proj proj = {});

    template<ForwardRange R, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp =
             ranges::less>
    constexpr safe_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<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr I upper_bound(I first, S last, const T& value, Comp comp = {},
                             Proj proj = {});

    template<ForwardRange R, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp =
             ranges::less>
    constexpr safe_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<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
             IndirectStrictWeakOrder<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<ForwardRange R, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp =
             ranges::less>
    constexpr safe_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);  
template<class ForwardIterator, class T, class Compare>
constexpr bool binary_search(ForwardIterator first, ForwardIterator last,
                             const T& value, Compare comp);

namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
    constexpr bool binary_search(I first, S last, const T& value, Comp comp = {},
                                  Proj proj = {});
    template<ForwardRange R, class T, class Proj = identity,
             IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp =
             ranges::less>
    constexpr bool binary_search(R&& r, const T& value, Comp comp = {},
                                  Proj proj = {});
}

// 25.7.4, 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, // see 25.3.5
                    ForwardIterator first, ForwardIterator last, Predicate pred);

namespace ranges {
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr bool is_partitioned(I first, S last, Pred pred, Proj proj = {});
    template<InputRange R, class Proj = identity,
             IndirectUnaryPredicate<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);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator partition(ExecutionPolicy&& exec, // see 25.3.5
                           ForwardIterator first,
                           ForwardIterator last,
                           Predicate pred);

namespace ranges {
    template<Permutable I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I partition(I first, S last, Pred pred, Proj proj = {});
}
template<ForwardRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>>, Proj>> Pred>
constexpr safe_iterator_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<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
        IndirectUnaryPredicate<projected<I>, Proj>> Pred>
    I stable_partition(I first, S last, Pred pred, Proj proj = {});
    template<BidirectionalRange R, class Proj = identity,
        IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
    safe_iterator_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>
    struct partition_copy_result {
        [[no_unique_address]] I in;
        [[no_unique_address]] O1 out1;
        [[no_unique_address]] O2 out2;
        template<class II, class OO1, class OO2>
            requiresConvertibleTo<const I&, II> &&
            ConvertibleTo<const O1&, OO1> && ConvertibleTo<const O2&, OO2>
            operator partition_copy_result<II, OO1, OO2>() const & {
                return {in, out1, out2};
            }
            template<class II, class OO1, class OO2>
            requiresConvertibleTo<II, I> &&
            ConvertibleTo<II, O1> && ConvertibleTo<II, O2>
            operator partition_copy_result<II, O1, O2>() && {
                return {std::move(in), std::move(out1), std::move(out2)};
            }
};
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable 01, WeaklyIncrementable 02,
        class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
constexpr partition_copy_result<I, 01, 02>
    partition_copy(I first, S last, 01 out_true, 02 out_false, Pred pred,
                         Proj proj = {});
template<InputRange R, WeaklyIncrementable 01, WeaklyIncrementable 02,
        class Proj = identity,
        IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
constexpr partition_copy_result<safe_iterator_t<R>, 01, 02>
    partition_copy(R&& r, 01 out_true, 02 out_false, Pred pred, Proj proj = {});
}

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last,
                        Predicate pred);

namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
            IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I partition_point(I first, S last, Pred pred, Proj proj = {});
    template<ForwardRange R, class Proj = identity,
             IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
    partition_point(R&& r, Pred pred, Proj proj = {});
}

// 25.7.5, merge
template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
    merge(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result);

namespace ranges {
    using merge_result = binary_transform_result<InputIterator1, InputIterator2, OutputIterator>;
}

§ 25.4
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, WeaklyIncrementable 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<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
  merge(R1&& r1, R2&& r2, O result,
        Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

} // namespace ranges

§ 25.4
namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
class Proj1 = identity, class Proj2 = identity,
IndirectStrictWeakOrder<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 = {});
}

namespace ranges {
  template<class InputIterator1, class InputIterator2, class OutputIterator
constexpr OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
}

namespace ranges {
  template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
class Proj1 = identity, class Proj2 = identity,
IndirectStrictWeakOrder<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 = {});
}

namespace ranges {
  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);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
ForwardIterator
    set_intersection(ExecutionPolicy&& exec, // see 25.3.5
                     ForwardIterator1 first1, ForwardIterator1 last1,
                     ForwardIterator2 first2, ForwardIterator2 last2,
                     ForwardIterator result);

namespace ranges {
    template<class I1, class I2, class O>
    using set_intersection_result = binary_transform_result<I1, I2, O>;

    template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
             WeaklyIncrementable 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<InputRange R1, InputRange R2, WeaklyIncrementable 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<safe_iterator_t<R1>, safe_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_difference(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, InputIterator2 last2,
                   OutputIterator result);

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);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
ForwardIterator
    set_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_difference(ExecutionPolicy&& exec, // see 25.3.5
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, ForwardIterator2 last2,
                   ForwardIterator result, Compare comp);
namespace ranges {
    template<class I, class O>
    using set_difference_result = copy_result<I, O>;

    template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
        WeaklyIncrementable O, class Comp = ranges::less,
        class Proj1 = identity, class Proj2 = identity>
    requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
    constexpr set_difference_result<I1, I2, O>
    set_difference(I1 first1, S1 last1, I2 first2, S2 last2, O result,
        Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

    template<InputRange R1, InputRange R2, WeaklyIncrementable 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_difference_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    set_difference(R1&& r1, R2&& r2, O result,
        Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});
}

namespace ranges {
    template<class InputIterator1, class InputIterator2, class OutputIterator>
    constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result);

    template<class InputIterator1, class InputIterator2, class OutputIterator, class Compare>
    constexpr OutputIterator
    set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result, Compare comp);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
        class ForwardIterator>
    ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        ForwardIterator result);

    template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
        class ForwardIterator, class Compare>
    ForwardIterator
    set_symmetric_difference(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        ForwardIterator result, Compare comp);
}

namespace ranges {
    template<class I1, class I2, class O>
    using set_symmetric_difference_result = binary_transform_result<I1, I2, O>;

    template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
        WeaklyIncrementable 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<InputRange R1, InputRange R2, WeaklyIncrementable 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<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    set_symmetric_difference(R1&& r1, R2&& r2, O result, Comp comp = {},
        Proj1 proj1 = {}, Proj2 proj2 = {});
}
// 25.7.7, heap operations

template<class RandomAccessIterator>
constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<class RandomAccessIterator>
constexpr void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<class RandomAccessIterator>
constexpr void make_heap(RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<class RandomAccessIterator>
constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

namespace ranges {
    template<RandomAccessIterator I, Sentinel<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 = {});
}
namespace ranges {
    template<RandomAccessIterator I, Sentinel<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<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr safe_iterator_t<R> sort_heap(R&& r, Comp comp = {}, Proj proj = {});
}

namespace ranges {
    template<class RandomAccessIterator>
    constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
    template<class RandomAccessIterator, class Compare>
    constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
    template<class ExecutionPolicy, class RandomAccessIterator>
    bool is_heap(ExecutionPolicy&& exec, // see 25.3.5
                 RandomAccessIterator first, RandomAccessIterator last);
    template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
    bool is_heap(ExecutionPolicy&& exec, // see 25.3.5
                 RandomAccessIterator first, RandomAccessIterator last, Compare comp);
}

namespace ranges {
    template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity, IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr bool is_heap(I first, S last, Comp comp = {}, Proj proj = {});
    template<RandomAccessRange R, class Proj = identity, IndirectStrictWeakOrder<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);
    template<class ExecutionPolicy, class RandomAccessIterator>
    RandomAccessIterator is_heap_until(ExecutionPolicy&& exec, // see 25.3.5
                                        RandomAccessIterator first, RandomAccessIterator last);
    template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
    RandomAccessIterator is_heap_until(ExecutionPolicy&& exec, // see 25.3.5
                                        RandomAccessIterator first, RandomAccessIterator last, Compare comp);
}

namespace ranges {
    template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity, IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I is_heap_until(I first, S last, Comp comp = {}, Proj proj = {});
    template<RandomAccessRange R, class Proj = identity, IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    constexpr safe_iterator_t<R> is_heap_until(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.7.8, minimum and maximum
template<class T> constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
constexpr const T& min(const T& a, const T& b, Compare comp);

template<class T>
constexpr T min(initializer_list<T> t);

template<class T, class Compare>
constexpr T min(initializer_list<T> t, Compare comp);

namespace ranges {
    template<class T, class Proj = identity,
             IndirectStrictWeakOrder<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,
          IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
constexpr T min(initializer_list<T> r, Comp comp = {}, Proj proj = {});

template<InputRange R, class Proj = identity,
          IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>>
constexpr iter_value_t<iterator_t<R>>
min(R&& r, Comp comp = {}, Proj proj = {});
}

template<class T> constexpr const T& max(const T& a, const T& b);

template<class T, class Compare>
constexpr const T& max(const T& a, const T& b, Compare comp);

template<class T>
constexpr T max(initializer_list<T> t);

template<class T, class Compare>
constexpr T max(initializer_list<T> t, Compare comp);

namespace ranges {
    template<class T, class Proj = identity,
             IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
    constexpr const T& max(const T& a, const T& b, Comp comp = {}, Proj proj = {});

template<Copyable T, class Proj = identity,
          IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
constexpr T max(initializer_list<T> r, Comp comp = {}, Proj proj = {});

template<InputRange R, class Proj = identity,
          IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>>
constexpr iter_value_t<iterator_t<R>>
max(R&& r, Comp comp = {}, Proj proj = {});
}

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);

template<class T, class Compare>
constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);

template<class T>
constexpr pair<T, T> minmax(initializer_list<T> t);

template<class T, class Compare>
constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);

namespace ranges {
    template<class T>
    struct minmax_result {
        [[no_unique_address]] T min;
        [[no_unique_address]] T max;

        template<class T2>
        requiresConvertibleTo<const T&, T2>
        operator minmax_result<T2>() const & {
            return {min, max};
        }
    }

§ 25.4 1023
template<class T2>
    requiresConvertibleTo<T, T2>
    operator minmax_result<T2>() && {
        return {std::move(min), std::move(max)};
    }
};

template<class T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
    constexpr minmax_result<const T&>
    minmax(const T& a, const T& b, Comp comp = {}, Proj proj = {});

template<Copyable T, class Proj = identity,
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
    constexpr minmax_result<T> minmax(initializer_list<T> r, Comp comp = {}, Proj proj = {});

template<InputRange R, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>
    requiresIndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>>
    constexpr minmax_result<iter_value_t<iterator_t<R>>>
    minmax(R&& r, Comp comp = {}, Proj proj = {});

namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
        IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I min_element(I first, S last, Comp comp = {}, Proj proj = {});
};

namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
        IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
    constexpr I max_element(I first, S last, Comp comp = {}, Proj proj = {});
};
template<ForwardRange R, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>>, Comp = ranges::less>
constexpr safe_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 = minmax_result<I>;

    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
        IndirectStrictWeakOrder<projected<I, Proj>>, Comp = ranges::less>
    constexpr minmax_element_result<I>
        minmax_element(I first, S last, Comp comp = {}, Proj proj = {});

template<ForwardRange R, class Proj = identity,
    IndirectStrictWeakOrder<projected<iterator_t<R>>, Proj>>, Comp = ranges::less>
    constexpr minmax_element_result<safe_iterator_t<R>>
        minmax_element(R&& r, Comp comp = {}, Proj proj = {});
}

// 25.7.9, bounded value
template<class T>
constexpr const T& clamp(const T& v, const T& lo, const T& hi);

template<class T, class Compare>
constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);

// 25.7.10, 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, // see 25.3.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
          class Compare>
bool
    lexicographical_compare(ExecutionPolicy&& exec, // see 25.3.5
                            ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2, Compare comp);
namespace ranges {
    template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
             class Proj1 = identity, class Proj2 = identity,
             IndirectStrictWeakOrder<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<InputRange R1, InputRange R2, class Proj1 = identity,
             class Proj2 = identity,
             IndirectStrictWeakOrder<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 = {});
}

// 25.7.11, three-way comparison algorithms

template<class T, class U>
constexpr auto compare_3way(const T& a, const U& b);

template<class InputIterator1, class InputIterator2, class Cmp>
constexpr auto lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
    InputIterator2 b2, InputIterator2 e2, Cmp comp)
    -> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>;

namespace ranges {
    template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
             class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool next_permutation(I first, S last, Comp comp = {}, Proj proj = {});
    template<BidirectionalRange R, class Comp = ranges::less,
             class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr bool next_permutation(R&& r, Comp comp = {}, Proj proj = {});
}

template<class BidirectionalIterator>
constexpr bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
constexpr bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last, Compare comp);

namespace ranges {
    template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
             class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool next_permutation(I first, S last, Comp comp = {}, Proj proj = {});
    template<BidirectionalRange R, class Comp = ranges::less,
             class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr bool next_permutation(R&& r, Comp comp = {}, Proj proj = {});
}

template<class BidirectionalIterator>
constexpr bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
constexpr bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last, Compare comp);

namespace ranges {
    template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
             class Proj = identity>
    requires Sortable<I, Comp, Proj>
    constexpr bool next_permutation(I first, S last, Comp comp = {}, Proj proj = {});
    template<BidirectionalRange R, class Comp = ranges::less,
             class Proj = identity>
    requires Sortable<iterator_t<R>, Comp, Proj>
    constexpr bool next_permutation(R&& r, Comp comp = {}, Proj proj = {});
}
25.5 Non-modifying sequence operations

25.5.1 All of

```cpp
template<class InputIterator, class Predicate>
constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool all_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, 
    Predicate pred);
```

```cpp
template<InputIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
constexpr bool ranges::all_of(I first, S last, Pred pred, Proj proj = {});
```

```cpp
template<InputRange R, class Proj = identity, 
    IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::all_of(R&& r, Pred pred, Proj proj = {});
```

1 Let \( E \) be \( \text{pred}(\ast\text{i}) \) and \( \text{invoke} \left( \text{pred}, \text{invoke} ( \text{proj}, \ast\text{i}) \right) \) for the overloads in namespace `std` and `std::ranges`, respectively.
2 Returns: `false` if \( E \) is `false` for some iterator \( i \) in the range \([\text{first}, \text{last})\), and `true` otherwise.
3 Complexity: At most `last - first` applications of the predicate and any projection.

25.5.2 Any of

```cpp
template<class InputIterator, class Predicate>
constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, 
    Predicate pred);
```

```cpp
template<InputIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
constexpr bool ranges::any_of(I first, S last, Pred pred, Proj proj = {});
```

```cpp
template<InputRange R, class Proj = identity, 
    IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::any_of(R&& r, Pred pred, Proj proj = {});
```

1 Let \( E \) be \( \text{pred}(\ast\text{i}) \) and \( \text{invoke} \left( \text{pred}, \text{invoke} ( \text{proj}, \ast\text{i}) \right) \) for the overloads in namespace `std` and `std::ranges`, respectively.
2 Returns: `true` if \( E \) is `true` for some iterator \( i \) in the range \([\text{first}, \text{last})\), and `false` otherwise.
3 Complexity: At most `last - first` applications of the predicate and any projection.

25.5.3 None of

```cpp
template<class InputIterator, class Predicate>
constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool none_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, 
    Predicate pred);
```

```cpp
template<InputIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
constexpr bool ranges::none_of(I first, S last, Pred pred, Proj proj = {});
```

```cpp
template<InputRange R, class Proj = identity, 
    IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::none_of(R&& r, Pred pred, Proj proj = {});
```

1 Let \( E \) be \( \text{pred}(\ast\text{i}) \) and \( \text{invoke} \left( \text{pred}, \text{invoke} ( \text{proj}, \ast\text{i}) \right) \) for the overloads in namespace `std` and `std::ranges`, respectively.
2 Returns: `true` if \( E \) is `true` for some iterator \( i \) in the range \([\text{first}, \text{last})\), and `false` otherwise.
3 Complexity: At most `last - first` applications of the predicate and any projection.
template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
constexpr bool ranges::none_of(R&& r, Pred pred, Proj proj = {});

Let E be pred(*i) and invoke(pred, invoke(proj, *i)) for the overloads in namespace std and std::ranges, respectively.

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.

25.5.4 For each

template<class InputIterator, class Function>
constexpr Function for_each(InputIterator first, InputIterator last, Function f);

Requires: Function shall satisfy the Cpp17MoveConstructible requirements (Table 26). [Note: Function need not meet the requirements of Cpp17CopyConstructible (Table 27). — end note]

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 satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator. — end note]

Returns: f.

Complexity: Applies f exactly last - first times.

Remarks: If f returns a result, the result is ignored.

[Note: Does not return a copy of its Function parameter, since parallelization may not permit efficient state accumulation. — end note]

template<class ExecutionPolicy, class ForwardIterator, class Function>
void for_each(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    Function f);

Requires: Function shall satisfy the Cpp17CopyConstructible requirements.

Effects: Applies f to the result of dereferencing every iterator in the range [first, last). [Note: If the type of first satisfies the requirements of a mutable iterator, f may apply non-constant functions through the dereferenced iterator. — end note]

Complexity: Applies f exactly last - first times.

Remarks: If f returns a result, the result is ignored. Implementations do not have the freedom granted under 25.3.3 to make arbitrary copies of elements from the input sequence.

[Note: Does not return a copy of its Function parameter, since parallelization may not permit efficient state accumulation. — end note]

template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryInvocable<projected<I, Proj>> Fun>
constexpr ranges::for_each_result<I, Fun>
ranges::for_each(I first, S last, Fun f, Proj proj = {});

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]

Returns: {last, std::move(f)}.

Complexity: Applies f and proj exactly last - first times.

Remarks: If f returns a result, the result is ignored.

[Note: The overloads in namespace ranges require Fun to model CopyConstructible. — end note]

template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);

Requires: Function shall satisfy the Cpp17MoveConstructible requirements [Note: Function need not meet the requirements of Cpp17CopyConstructible. — end note]
Requires: \( n \geq 0 \).

Effects: Applies \( f \) to the result of dereferencing every iterator in the range \( [\text{first}, \text{first} + n) \) in order. [Note: If the type of \( \text{first} \) satisfies the requirements of a mutable iterator, \( f \) may apply non-constant functions through the dereferenced iterator. — end note]

Returns: \( \text{first} + n \).

Remarks: If \( f \) returns a result, the result is ignored.

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
ForwardIterator for_each_n(ExecutionPolicy&& exec, ForwardIterator first, Size n, Function f);
```

Requires: Function shall satisfy the \textit{Cpp17CopyConstructible} requirements.

Effects: Applies \( f \) to the result of dereferencing every iterator in the range \( [\text{first}, \text{first} + n) \). [Note: If the type of \( \text{first} \) satisfies the requirements of a mutable iterator, \( f \) may apply non-constant functions through the dereferenced iterator. — end note]

Returns: \( \text{first} + n \).

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.

### 25.5.5 Find

[alg.find]
template<InputRange R, class Proj = identity, 
  IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred> 
constexpr safe_iterator_t<R> 
ranges::find_if_not(R&& r, Pred pred, Proj proj = {});

1. Let $E$ be:
   
   - (1.1) $i = \text{value for find},$
   - (1.2) $\text{pred}(i) \neq \text{false for find_if},$
   - (1.3) $\text{pred}(i) = \text{false for find_if_not},$
   - (1.4) $\text{invoke(proj, } i) = \text{value for ranges::find},$
   - (1.5) $\text{invoke(pred, invoke(proj, } i)) \neq \text{false for ranges::find_if},$
   - (1.6) $\text{invoke(pred, invoke(proj, } i)) = \text{false for ranges::find_if_not}.$

2. Returns: The first iterator $i$ in the range $[\text{first, last})$ for which $E$ is true. Returns last if no such iterator is found.

3. Complexity: At most last - first applications of the corresponding predicate and any projection.

25.5.6 Find end [alg.find.end]

template<class ForwardIterator1, class ForwardIterator2> 
constexpr ForwardIterator1 
find_end(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2> 
ForwardIterator1 
find_end(ExecutionPolicy&& exec, 
ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, 
class BinaryPredicate> 
constexpr ForwardIterator1 
find_end(ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2, 
BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
class BinaryPredicate> 
ForwardIterator1 
find_end(ExecutionPolicy&& exec, 
ForwardIterator1 first1, ForwardIterator1 last1, 
ForwardIterator2 first2, ForwardIterator2 last2, 
BinaryPredicate pred);

template<class ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2, 
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity> 
requires IndirectlyComparable<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<class ForwardRange R1, ForwardRange R2, 
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity> 
requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2> 
constexpr safe_subrange_t<R1> 
ranges::find_end(R1&& r1, R2&& r2, Pred pred = {}, 
Proj1 proj1 = {}, Proj2 proj2 = {});

1. Let:
   
   - (1.1) $\text{pred be equal_to{}}$ for the overloads with no parameter \text{pred}.
   - (1.2) $E$ be:
     - (1.2.1) $\text{pred}((i + n), *(\text{first2} + n))$ for the overloads in namespace \text{std},
— invoke(pred, invoke(proj1, *(i + n)), invoke(proj2, *(first2 + n))) for the overloads in namespace ranges.

— i be last1 if [first2, last2) is empty, or if (last2 - first2) > (last1 - first1) is true, or if there is no iterator in the range [first1, last1 - (last2 - first2)) such that for every non-negative integer n < (last2 - first2), \( E \) is true. Otherwise i is the last such iterator in [first1, last1 - (last2 - first2)).

Returns:
— i for the overloads in namespace std, and
— \{i, i + (i == last1 ? 0 : last2 - first2)\} for the overloads in namespace ranges.

Complexity: At most \((last2 - first2) \ast (last1 - first1 - (last2 - first2) + 1)\) applications of the corresponding predicate and any projections.

25.5.7 Find first

§ 25.5.7 1031
2 Effects: Finds an element that matches one of a set of values.
3 Returns: The first iterator \( i \) in the range \([first1, last1)\) such that for some iterator \( j \) in the range \([first2, last2)\) \( E \) holds. Returns \( \text{last1} \) if \([first2, last2)\) is empty or if no such iterator is found.
4 Complexity: At most \((\text{last1}-\text{first1}) \times (\text{last2}-\text{first2})\) applications of the corresponding predicate and any projections.

25.5.8 Adjacent find

```cpp
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);
```

```cpp
// Additional functions
```

Let \( E \) be:

1. \( i == *(i + 1) \) for the overloads with no parameter \( \text{pred} \),
2. \( \text{pred}(*i, *(i + 1)) \neq \text{false} \) for the overloads with a parameter \( \text{pred} \) and no parameter \( \text{proj} \),
3. \( \text{invoke}(<\text{pred}, \text{invoke}(<\text{proj}, *i), \text{invoke}(<\text{proj}, *(i + 1))>) \neq \text{false} \) for the overloads with both parameters \( \text{pred} \) and \( \text{proj} \).

Returns: The first iterator \( i \) such that both \( i \) and \( i + 1 \) are in the range \([\text{first}, \text{last})\) for which \( E \) holds. Returns \( \text{last} \) if no such iterator is found.

Complexity: For the overloads with no \( \text{ExecutionPolicy} \), exactly

\[
\min((\text{last} - \text{first}) + 1, (\text{last} - \text{first}) - 1)
\]

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.

25.5.9 Count

```cpp
template<class InputIterator, class T>
constexpr typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
typename iterator_traits<ForwardIterator>::difference_type
count(ExecutionPolicy&& exec,
      ForwardIterator first, ForwardIterator last, const T& value);
```

```cpp
// Additional functions
```
template<class ExecutionPolicy, class ForwardIterator, class Predicate>

count_if(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last, Predicate pred);

template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
requires IndirectRelation::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<InputRange R, class T, class Proj = identity>
requires IndirectRelation::ranges::equal_to, projected<iterator_t<R>, Proj>, const T**
constexpr iter_difference_t<iterator_t<R>> ranges::count(R&& r, const T& value, Proj proj = {});

template<InputIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<Projected<I, Proj>> Pred>
constexpr iter_difference_t<I> ranges::count_if(I first, S last, Pred pred, Proj proj = {});

template<InputRange R, class Proj = identity,
    IndirectUnaryPredicate<Projected<iterator_t<R>, Proj>> Pred>
constexpr iter_difference_t<iterator_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. \( \text{pred}(*i) \neq \text{false} \) 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. \( \text{invoke} (\text{pred, invoke} (\text{proj}, *i)) \neq \text{false} \) 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.

### 25.5.10 Mismatch

```cpp
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>
    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<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, 
        class Proj1 = identity, class Proj2 = identity, 
        IndirectRelation<projected<I1, Proj1>, 
        projected<I2, Proj2>> Pred = ranges::equal_to>
   constexpr ranges::mismatch_result<I1, I2> mismatch(I1 first1, S1 last1, I2 first2, S2 last2, 
        Pred pred = {}, 
        Proj1 proj1 = {}, Proj2 proj2 = {});

template<InputRange R1, InputRange R2, 
        class Proj1 = identity, class Proj2 = identity, 
        IndirectRelation<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Pred = ranges::equal_to>
    constexpr ranges::mismatch_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
        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:
   (2.1) — !(*(first1 + n) == *(first2 + n)) for the overloads with no parameter pred,
   (2.2) — pred(*(first1 + n), *(first2 + n)) == false for the overloads with a parameter pred and
            no parameter proj1,
   (2.3) — !invoke(pred, invoke(proj1, *(first1 + n)), invoke(proj2, *(first2 + n))) for the
            overloads with both parameters pred and 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.

25.5.11 Equal

section

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, BinaryPredicate pred);

template<class InputIterator1, class InputIterator2>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
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);

template<class InputIterator1 I1, Sentinel<I1> S1, InputIterator1 I2, Sentinel<I2> S2,
class Pred = ranges::equal_to, class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
constexpr bool ranges::equal(I1 first1, S1 last1, I2 first2, S2 last2,
Pred pred = {},
Proj1 proj1 = {}, Proj2 proj2 = {});

template<InputRange R1, InputRange R2, class Pred = ranges::equal_to,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
constexpr bool ranges::equal(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,
   — pred be equal_to{} for the overloads with no parameter pred,
   — E be:
     — pred(*i, *(first2 + (i - first1))) for the overloads with no parameter proj1,
     — invoke(pred, invoke(proj1, *i), invoke(proj2, *(first2 + (i - first1)))) for
       the overloads with parameter proj1.

2 Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if E holds
   for every iterator i in the range [first1, last1) Otherwise, returns false.

3 Complexity: If the types of first1, last1, first2, and last2:
   — meet the Cpp17RandomAccessIterator requirements (23.3.5.6) for the overloads in namespace std,
   or
   — pairwise model SizedSentinel (23.3.4.8) for the overloads in namespace ranges,
   and last1 - first1 != last2 - first2, then no applications of the corresponding predicate and
each projection; otherwise,
   — For the overloads with no ExecutionPolicy, at most min(last1 - first1, last2 - 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.
25.5.12 Is permutation [alg.is.permutation]

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);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);

Requires: ForwardIterator1 and ForwardIterator2 shall have the same value type. The comparison function shall be 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<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2,
Sentinel<I2> S2, class Pred = ranges::equal_to, class Proj1 = identity,
class Proj2 = identity>
requires IndirectlyComparable<I1, I2, Pred, Proj1, Proj2>
constexpr bool ranges::is_permutation(I1 first1, S1 last1, I2 first2, S2 last2,
Pred pred = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

template<ForwardRange R1, ForwardRange R2, class Pred = ranges::equal_to,
class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
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 SizedSentinel,
2. S2 and I2 model SizedSentinel, 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.

25.5.13 Search [alg.search]

template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator1

§ 25.5.13 1036
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: \(*(i + n) == *(first2 + n)\) and \(\text{pred}(*(i + n), *(first2 + n)) \neq false\). Returns \(first1\) if \([first2, last2)\) is empty, otherwise returns \(last1\) if no such iterator is found.

2 Complexity: At most \((last1 - first1) * (last2 - first2)\) applications of the corresponding predicate.

template<ForwardIterator I1, Sentinel<I1> S1, ForwardIterator I2, Sentinel<I2> S2, class Pred = ranges::equal_to,
            class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<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<ForwardRange R1, ForwardRange R2, class Pred = ranges::equal_to,
           class Proj1 = identity, class Proj2 = identity>
requires IndirectlyComparable<iterator_t<R1>, iterator_t<R2>, Pred, Proj1, Proj2>
constexpr safe_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
\[
\text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj1}, *(i + n)), \text{invoke}(\text{proj2}, *(first2 + n))))
\]
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.

template<class ForwardIterator, class Size, class T>
calculate 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);

template<class ExecutionPolicy, class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator
search_n(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    Size count, const T& value,
    BinaryPredicate pred);

**Requires:** The type Size shall be convertible to integral type (7.3.8, 11.3.7).

**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<class ForwardIterator I, Sentinel<I> S, class T, class Pred = ranges::equal_to, class Proj = identity>
requires IndirectlyComparable<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<ForwardRange R, class T, class Pred = ranges::equal_to, class Proj = identity>
requires IndirectlyComparable<iterator_t<R>, const T*, Pred, Proj>
constexpr safe_subrange_t<R>
ranges::search_n(R&& r, iter_difference_t<iterator_t<R>> count,
    const T& value, Pred pred = {}, Proj proj = {});

**Returns:** {i, i + count} where i is the first iterator in the range [first, last - count) such that for every non-negative integer n less than count, the following condition holds: invoke(pred, invoke(proj, *(i + n)), value). Returns {last, last} if no such iterator is found.

**Complexity:** At most last - first applications of the corresponding predicate and projection.

template<class ForwardIterator, class Searcher>
constexpr ForwardIterator
search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);

**Effects:** Equivalent to: return searcher(first, last).first;

**Remarks:** Searcher need not meet the Cpp17CopyConstructible requirements.

### 25.6 Mutating sequence operations

#### 25.6.1 Copy

template<class InputIterator, class OutputIterator>
constexpr OutputIterator
copy(InputIterator first, InputIterator last,
    OutputIterator result);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
requires IndirectlyCopyable<I, O>
constexpr ranges::copy_result<I, O> ranges::copy(I first, S last, O result);

template<InputRange R, WeaklyIncrementable O>
requires IndirectlyCopyable<iterator_t<R>, O>
constexpr ranges::copy_result<safe_iterator_t<R>, O> ranges::copy(R&& r, O result);

Let N be last - first.

**Requires:** result shall not be in the range [first, last).
Effects: Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + N)\) starting from \text{first} and proceeding to \text{last}. For each non-negative integer \(n < N\), performs \(\*(\text{result} + n) = \*(\text{first} + n)\).

Returns:
\[
\begin{align*}
&\quad \text{result} + N \text{ for the overload in namespace std, or} \\
&\quad \{\text{last}, \text{result} + N\} \text{ for the overloads in namespace ranges.}
\end{align*}
\]

Complexity: Exactly \(N\) assignments.

```
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 copy(ExecutionPolicy&& policy,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result);
```

Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

Effects: Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\). For each non-negative integer \(n < (\text{last} - \text{first})\), performs \(\*(\text{result} + n) = \*(\text{first} + n)\).

Returns: \(\text{result} + (\text{last} - \text{first})\).

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<InputIterator I, WeaklyIncrementable O>
requires IndirectlyCopyable<I, O>
constexpr ranges::copy_n_result<I, O>
ranges::copy_n(I first, iter_difference_t<I> n, O result);
```

Let \(M\) be \(\text{max}(n, 0)\).

Effects: For each non-negative integer \(i < M\), performs \(\*(\text{result} + i) = \*(\text{first} + i)\).

Returns:
\[
\begin{align*}
&\quad \text{result} + M \text{ for the overloads in namespace std, or} \\
&\quad \{\text{first} + M, \text{result} + M\} \text{ for the overload in namespace ranges.}
\end{align*}
\]

Complexity: Exactly \(M\) 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<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
requires IndirectlyCopyable<I, O>
constexpr ranges::copy_if_result<I, O>
ranges::copy_if(I first, S last, O result, Pred pred, Proj proj = {});
```

§ 25.6.1
ranges::copy_if(R&& r, O result, Pred pred, Proj proj = {});

Let \( E \) be:

\begin{align*}
(14.1) & \quad \text{— } bool(\text{pred}(\ast i)) \text{ for the overloads in namespace \texttt{std}, or} \\
(14.2) & \quad \text{— } bool(\text{invoke}(\text{pred, invoke}(\text{proj, } \ast i))) \text{ for the overloads in namespace \texttt{ranges}}.
\end{align*}

and \( N \) be the number of iterators \( i \) in the range \([\text{first, last})\) for which the condition \( E \) holds.

**Requires:** The ranges \([\text{first, last})\) and \([\text{result, result + (last - first)})\) shall not overlap.

[Note: For the overload with an \texttt{ExecutionPolicy}, there may be a performance cost if \texttt{iterator_traits<ForwardIterator>::value_type} is not \texttt{Cpp17MoveConstructible} (Table 26). — end note]

**Effects:** Copies all of the elements referred to by the iterator \( i \) in the range \([\text{first, last})\) for which \( E \) is true.

**Returns:**

\begin{align*}
(17.1) & \quad \text{— } \text{result} + N \text{ for the overloads in namespace \texttt{std}, or} \\
(17.2) & \quad \{\text{last, result} + N\} \text{ for the overloads in namespace \texttt{ranges}}.
\end{align*}

**Complexity:** Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate and any projection.

**Remarks:** Stable (16.5.5.7).

\begin{verbatim}
template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
    copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
                  BidirectionalIterator2 result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
requires IndirectlyCopyable<I1, I2>
constexpr ranges::copy_backward_result<I1, I2>
    ranges::copy_backward(I1 first, S1 last, I2 result); 

template<BidirectionalRange R, BidirectionalIterator I>
requires IndirectlyCopyable<iterator_t<R>, I>
constexpr ranges::copy_backward_result<safe_iterator_t<R>, I>
    ranges::copy_backward(R&& r, I result);
\end{verbatim}

Let \( N \) be \( \text{last} - \text{first} \).

**Requires:** \texttt{result} shall not be in the range \([\text{first, last})\).

**Effects:** Copies elements in the range \([\text{first, last})\) into the range \([\text{result} - N, \text{result})\) starting from \( \text{last} - 1 \) and proceeding to \( \text{first} \).\(^{235}\) For each positive integer \( n \leq N \), performs \( *(\text{result} - n) = *(\text{last} - n) \).

**Returns:**

\begin{align*}
(23.1) & \quad \text{— } \text{result} - N \text{ for the overload in namespace \texttt{std}, or} \\
(23.2) & \quad \{\text{last, result} - N\} \text{ for the overloads in namespace \texttt{ranges}}.
\end{align*}

**Complexity:** Exactly \( N \) assignments.

\subsection*{25.6.2 Move [alg.move]}

\begin{verbatim}
template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last, OutputIterator result);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O>
requires IndirectlyMovable<I, O>
constexpr ranges::move_result<I, O>
    ranges::move(I first, S last, O result);
\end{verbatim}

\(^{235}\) \texttt{copy_backward} should be used instead of \texttt{copy} when \( \text{last} \) is in the range \([\text{result} - N, \text{result})\).
template<InputRange R, WeaklyIncrementable O>
requires IndirectlyMovable<iterator_t<R>, O>
constexpr ranges::move_result<safe_iterator_t<R>, O>
ranges::move(R&& r, O result);

Let \( E \) be

\begin{align}
(1.1) & \quad \text{std::move(*}(first + n)) \text{ for the overload in namespace std, or} \\
(1.2) & \quad \text{ranges::iter_move(first + n) for the overloads in namespace ranges.}
\end{align}

Let \( N \) be \( last - first \).

**Requires:** \( result \) shall not be in the range \( [first, last) \).

**Effects:** Moves elements in the range \( [first, last) \) into the range \( [result, result + N) \) starting from \( first \) and proceeding to \( last \). For each non-negative integer \( n < N \), performs \( *(result + n) = E \).

**Returns:**

\begin{align}
(4.1) & \quad result + N \text{ for the overload in namespace std, or} \\
(4.2) & \quad \{last, result + N\} \text{ for the overloads in namespace ranges.}
\end{align}

**Complexity:** Exactly \( N \) assignments.

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 move(ExecutionPolicy&& policy, ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);

Let \( N \) be \( last - first \).

**Requires:** The ranges \( [first, last) \) and \( [result, result + N) \) shall not overlap.

**Effects:** Moves elements in the range \( [first, last) \) into the range \( [result, result + N) \). For each non-negative integer \( n < N \), performs \( *(result + n) = \text{std::move(*}(first + n)) \).

**Returns:** \( result + N \).

**Complexity:** Exactly \( N \) assignments.

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2 move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last, BidirectionalIterator2 result);

template<BidirectionalIterator I1, Sentinel<I1> S1, BidirectionalIterator I2>
requires IndirectlyMovable<I1, I2>
constexpr ranges::move_backward_result<I1, I2>
ranges::move_backward(I1 first, S1 last, I2 result);

template<BidirectionalRange R, BidirectionalIterator I>
requires IndirectlyMovable<iterator_t<R>, I>
constexpr ranges::move_backward_result<safe_iterator_t<R>, I>
ranges::move_backward(R&& r, I result);

Let \( E \) be

\begin{align}
(11.1) & \quad \text{std::move(*}(last - n)) \text{ for the overload in namespace std, or} \\
(11.2) & \quad \text{ranges::iter_move(last - n) for the overloads in namespace ranges.}
\end{align}

Let \( N \) be \( last - first \).

**Requires:** \( result \) shall not be in the range \( [first, last) \).

**Effects:** Moves elements in the range \( [first, last) \) into the range \( [result - N, result) \) starting from \( last - 1 \) and proceeding to \( first \). For each positive integer \( n \leq N \), performs \( *(result - n) = E \).

**Returns:**

\begin{align}
(14.1) & \quad result - N \text{ for the overload in namespace std, or}
\end{align}

---

\[^{236}]\text{move_backward should be used instead of move when last is in the range [result - N, result].}
25.6.3 Swap

```cpp
template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
swap_ranges(ExecutionPolicy&& exec,
            ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2>
requires IndirectlySwappable<I1, I2>
ranges::swap_ranges_result<I1, I2>
ranges::swap_ranges(I1 first1, S1 last1, I2 first2, S2 last2);

template<InputRange R1, InputRange R2>
requires IndirectlySwappable<iterator_t<R1>, iterator_t<R2>>
ranges::swap_ranges_result<safe_iterator_t<R1>, safe_iterator_t<R2>>
ranges::swap_ranges(R1&& r1, R2&& r2);
```

Let:

1. last2 be \( \text{first2 + (last1 - first1)} \) for the overloads with no parameter named \( \text{last2} \), and
2. \( M \) be \( \min(\text{last1 - first1}, \text{last2 - first2}) \).

**Requires:** The two ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) shall not overlap. For the overloads in namespace std, \(*(\text{first1 + n})\) shall be swappable with \((16.5.3.2) * (\text{first2 + n})\).

**Effects:** For each non-negative integer \( n < M \) performs:

1. \( \text{swap}(*(\text{first1 + n}), *(\text{first2 + n})) \) for the overloads in namespace std, or
2. \( \text{ranges::iter_swap(}\text{first1 + n, first2 + n}) \) for the overloads in namespace ranges.

**Returns:**

1. last2 for the overloads in namespace std, or
2. \( \{\text{first1 + M, first2 + M}\} \) for the overloads in namespace ranges.

**Complexity:** Exactly \( M \) swaps.

25.6.4 Transform

```cpp
template<class InputIterator, class OutputIterator,
         class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first1, InputIterator last1,
          OutputIterator result, UnaryOperation op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class UnaryOperation>
ForwardIterator2
transform(ExecutionPolicy&& exec,
          ForwardIterator1 first1, ForwardIterator1 last1,
          ForwardIterator2 result, UnaryOperation op);
```
template<class InputIterator1, class InputIterator2,
    class OutputIterator, class BinaryOperation>
constexpr OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, OutputIterator result,
    BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class ForwardIterator, class BinaryOperation>
ForwardIterator
transform(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator result,
    BinaryOperation binary_op);

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
    CopyConstructible F, class Proj = identity>
requires 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<InputRange R, WeaklyIncrementable O, CopyConstructible F,
    class Proj = identity>
requires Writable<O, indirect_result_t<F&, projected<iterator_t<R>, Proj>>
constexpr ranges::unary_transform_result<safe_iterator_t<R>, O>
ranges::transform(R&& r, O result, F op, Proj proj = {});

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, CopyConstructible F, class Proj1 = identity,
    class Proj2 = identity>
requires Writable<O, indirect_result_t<F&, projected<iterator_t<R1>, Proj1>,
    projected<iterator_t<R2>, Proj2>>
constexpr ranges::binary_transform_result<I1, I2, O>
ranges::transform(I1 first1, S1 last1, I2 first2, S2 last2, O result,
    F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});

template<InputRange R1, InputRange R2, WeaklyIncrementable O,
    CopyConstructible F, class Proj1 = identity, class Proj2 = identity>
requires Writable<O, indirect_result_t<F&, projected<iterator_t<R1>, Proj1>,
    projected<iterator_t<R2>, Proj2>>
constexpr ranges::binary_transform_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
ranges::transform(R1&& r1, R2&& r2, O result,
    F binary_op, Proj1 proj1 = {}, Proj2 proj2 = {});

Let:

1

Let:

(1.1) last2 be first2 + (last1 - first1) for the overloads with parameter first2 but no parameter last2,

(1.2) N be last1 - first1 for unary transforms, or min(last1 - first1, last2 - first2) for binary transforms, and

(1.3) E be

1 op(*(first1 + (i - result))) for unary transforms defined in namespace std,

1.1.2 binary_op(*(first1 + (i - result)), *(first2 + (i - result))) for binary transforms defined in namespace std,

1.1.3 invoke(op, invoke(proj, *(first1 + (i - result)))) for unary transforms defined in namespace ranges, or

1.1.4 invoke(binary_op, invoke(proj1, *(first1 + (i - result))), invoke(proj2, *(first2 + (i - result)))) for binary transforms defined in namespace ranges.

2 Requires: op and binary_op shall not invalidate iterators or subranges, nor modify elements in the ranges

2.1 [first1, first1 + N],

2.2 [first2, first2 + N], and
Let $E$ be

\begin{align}
(1.1) & \quad \text{bool}(*i == \text{old\_value}) \text{ for replace}, \\
(1.2) & \quad \text{bool}(	ext{pred}(*i)) \text{ for replace\_if}, \\
(1.3) & \quad \text{bool}(\text{invoke}(	ext{proj}, *i) == \text{old\_value}) \text{ for } \text{ranges}::\text{replace}, \text{ or} \\
(1.4) & \quad \text{bool}(\text{invoke}(	ext{pred}, \text{invoke}(	ext{proj}, *i))) \text{ for } \text{ranges}::\text{replace\_if}.
\end{align}

\footnote{The use of fully closed ranges is intentional.}
Requires: The expression `*first = new_value` shall be valid.

Effects: Substitutes elements referred by the iterator `i` in the range `[first, 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<InputIterator I, Sentinel<I> S, class T1, class T2, OutputIterator<const T2&> O,
class Proj = identity>
requires IndirectlyCopyable<I, O> &&
IndirectRelation<ranges::equal_to, projected<I, Proj>, const T1*>,
constexpr ranges::replace_copy(I first, S last, O result, const T1& old_value, const T2& new_value,
Proj proj = {});

template<InputRange R, class T1, class T2, OutputIterator<const T2&> O,
class Proj = identity>
requires IndirectlyCopyable<iterator_t<R>, O> &&
IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T1*>,
constexpr ranges::replace_copy(R&& r, O result, const T1& old_value, const T2& new_value,
Proj proj = {});

template<InputIterator I, Sentinel<I> S, class T, OutputIterator<const T&> O,
class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
requires IndirectlyCopyable<I, O> &&
IndirectRelation<ranges::equal_to, projected<I, Proj>, Pred>,
constexpr ranges::replace_copy_if(I first, S last, O result, Pred pred, const T& new_value,
Proj proj = {});

template<InputRange R, class T, OutputIterator<const T&> O, class Proj = identity,
IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
requires IndirectlyCopyable<iterator_t<R>, O> &&
IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, Pred>,
constexpr ranges::replace_copy_if(R&& r, O result, Pred pred, const T& new_value,
Proj proj = {});

Let `E` be

(6.1)  \[ \text{bool}(\ast (\text{first} + (i - \text{result})) == \text{old_value}) \text{ for replace_copy}, \]

(6.2)  \[ \text{bool}(	ext{pred}((\ast (\text{first} + (i - \text{result})))) \text{ for replace_copy_if}, \]

§ 25.6.5
— `bool(invoke(proj, *(first + (i - result))) == old_value)` for `ranges::replace_copy`,

— `bool(invoke(pred, invoke(proj, *(first + (i - result)))) )` for `ranges::replace_copy_if`.

**Requires:** The results of the expressions `*first` and `new_value` shall be writable (23.3.1) to the result output iterator. The ranges `[first, last)` and `[result, result + (last - first))` shall not overlap.

**Effects:** Assigns to every iterator `i` in the range `[result, result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether `E` holds.

**Returns:**

1. `result + (last - first)` for the overloads in namespace `std`, or
2. `{last, result + (last - first)}` for the overloads in namespace `ranges`.

**Complexity:** Exactly `last - first` applications of the corresponding predicate and any projection.

### 25.6.6 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);

template<class OutputIterator, class Size, class T>
constexpr 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);
```

1. Let `N` be `max(0, n)` for the `fill_n` algorithms, and `last - first` for the `fill` algorithms.
2. **Requires:** The expression `value` shall be writable (23.3.1) to the output iterator. The type `Size` shall be convertible to an integral type (7.3.8, 11.3.7).
3. **Effects:** Assigns `value` through all the iterators in the range `[first, first + N)`.
4. **Returns:** `first + N`.
5. **Complexity:** Exactly `N` assignments.

### 25.6.7 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);

template<class OutputIterator, class Size, class Generator>
constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
ForwardIterator generate_n(ExecutionPolicy&& exec,
                           ForwardIterator first, Size n, Generator gen);
```

§ 25.6.7
template<Iterator O, Sentinel<O> S, CopyConstructible F>
  requires Invocable<F&> && Writable<O, invoke_result_t<F&>>
  constexpr O ranges::generate(O first, S last, F gen);

template<class R, CopyConstructible F>
  requires Invocable<F&> && OutputRange<R, invoke_result_t<F&>>
  constexpr safe_iterator_t<R> ranges::generate(R&& r, F gen);

template<Iterator O, CopyConstructible F>
  requires Invocable<F&> && 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} \_\texttt{N} algorithms, and \( \text{last} - \text{first} \) for the \texttt{generate} algorithms.

\textbf{Requires:} Size shall be convertible to an integral type (7.3.8, 11.3.7).

\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.

\subsection*{25.6.8 Remove} [\texttt{alg.remove}]

\begin{verbatim}
template<class ForwardIterator, class T>
  constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                  const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator remove(ExecutionPolicy&& exec,
                         ForwardIterator first, ForwardIterator last,
                         const T& value);

template<class ForwardIterator, class Predicate>
  constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                      Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ExecutionPolicy&& exec,
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);

template<Permutable I, Sentinel<I> S, class T, class Proj = identity>
  requires IndirectRelation<ranges::equal_to, projected<I, Proj>, const T*>
  constexpr I ranges::remove(I first, S last, const T& value, Proj proj = {});

template<ForwardRange R, class T, class Proj = identity>
  requires Permutable<iterator_t<R>> &&
              IndirectRelation<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*> 
  constexpr safe_iterator_t<R>
  ranges::remove(R&& r, const T& value, Proj proj = {});

template<Permutable I, Sentinel<I> S, class T, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred>
  requires Permutable<iterator_t<R>>
  constexpr safe_iterator_t<R>
  ranges::remove_if(I first, S last, Pred pred, Proj proj = {});

template<ForwardRange R, class T, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
  requires Permutable<iterator_t<R>>
  constexpr safe_iterator_t<R>
  ranges::remove_if(R&& r, Pred pred, Proj proj = {});
\end{verbatim}

Let \( E \) be

\begin{enumerate}
\item \( \text{bool}(\ast i == \text{value}) \) for \texttt{remove},
\item \( \text{bool}((\text{pred})(\ast i)) \) for \texttt{remove_if},
\item \( \text{bool}(\text{invoke}(\text{proj}, \ast i) == \text{value}) \) for \texttt{ranges::remove}, or
\item \( \text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, \ast i))) \) for \texttt{ranges::remove_if}.
\end{enumerate}

\textbf{Requires:} For the algorithms in namespace \texttt{std}, the type of \*\texttt{first} shall meet the \texttt{Cpp17MoveAssignabe} requirements (Table 28).
**Effects:** Eliminates all the elements referred to by iterator \( i \) in the range \([\text{first}, \text{last})\) for which \( E \) holds.

**Returns:** The end of the resulting range.

**Remarks:** Stable [16.5.5.7].

**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<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class T,
           class Proj = identity>
requires IndirectlyCopyable<I, O> &&
IndirectRelation<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<InputRange R, WeaklyIncrementable O, class T, class Proj = identity>
requires IndirectlyCopyable<iterator_t<R>, O> &&
IndirectRelation<ranges::equal_to, projected<iterator_t<R>>, Proj>, const T**>
constexpr ranges::remove_copy_result<safe_iterator_t<R>, O>
ranges::remove_copy(R&& r, O result, const T& value, Proj proj = {});

template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O,
           class Proj = identity, IndirectUnaryPredicate<projected<I>, Proj>> Pred>
requires IndirectlyCopyable<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<InputRange R, WeaklyIncrementable O, class Proj = identity,
          IndirectUnaryPredicate<projected<iterator_t<R>>, Proj>> Pred>
requires IndirectlyCopyable<iterator_t<R>, O>
constexpr ranges::remove_copy_if_result<safe_iterator_t<R>, O>
ranges::remove_copy_if(R&& r, O result, Pred pred, Proj proj = {});

Let \( E \) be

1. \( \text{bool}(\*i == \text{value}) \) for remove_copy,
2. \( \text{bool}(\text{pred}(\*i)) \) for remove_copy_if,
3. \( \text{bool}(\text{invoke(proj, } \*i) == \text{value}) \) for ranges::remove_copy, or
4. \( \text{bool}(\text{invoke(pred, invoke(proj, } \*i))) \) for ranges::remove_copy_if.

Let \( N \) be the number of elements in \([\text{first}, \text{last})\) for which \( E \) is false.
Requires: The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap. The expression \(*\text{result} = *\text{first}\) shall be valid. [Note: For the overloads with an ExecutionPolicy, there may be a performance cost if \text{iterator_traits<ForwardIterator1>::value_type} does not meet the \text{Cpp17MoveConstructible} (Table 26) 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:

(12.1) \(- \text{result} + N\), for the algorithms in namespace std, or

(12.2) \{-\text{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 (16.5.5.7).

25.6.9 Unique

\[
\text{template<class ForwardIterator>}
\text{constexpr ForwardIterator unique(ForwardIterator \text{first}, ForwardIterator \text{last});}
\]

\[
\text{template<class ExecutionPolicy, class ForwardIterator>}
\text{ForwardIterator unique(ExecutionPolicy&\& \text{exec},}
\text{ForwardIterator \text{first}, ForwardIterator \text{last});}
\]

\[
\text{template<class ForwardIterator, class BinaryPredicate>}
\text{constexpr ForwardIterator unique(ForwardIterator \text{first}, ForwardIterator \text{last},}
\text{BinaryPredicate \text{pred});}
\]

\[
\text{template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>}
\text{ForwardIterator unique(ExecutionPolicy&\& \text{exec},}
\text{ForwardIterator \text{first}, ForwardIterator \text{last},}
\text{BinaryPredicate \text{pred});}
\]

\[
\text{template<Permutable I, Sentinel<I> S, class Proj = identity,}
\text{IndirectRelation<projected<I, Proj>> C = ranges::equal_to<>}
\text{constexpr I ranges::unique(I \text{first}, S \text{last}, C \text{comp} = \{\}, \text{Proj \text{proj} = \{\});}
\]

\[
\text{template<ForwardRange R, class Proj = identity,}
\text{IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>}
\text{requires Permutable<iterator_t<R>>}
\text{constexpr safe_iterator_t<R>}
\text{ranges::unique(R&\& \text{r}, C \text{comp} = \{\}, \text{Proj \text{proj} = \{\});}
\]

1 Let \(\text{pred}\) be \text{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, or

(1.2) \(- \text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *(i - 1)), \text{invoke}(\text{proj, } *i)))\) for the overloads in namespace ranges.

2 Requires: For the overloads in namespace std, \(\text{pred}\) shall be an equivalence relation and the type of \(*\text{first}\) shall meet the \text{Cpp17MoveAssignable} requirements (Table 28).

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: The end of the resulting range.

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.

\[
\text{template<class InputIterator, class OutputIterator>}
\text{constexpr OutputIterator}
\text{unique_copy(InputIterator \text{first}, InputIterator \text{last},}
\text{OutputIterator \text{result});}
\]

\[
\text{template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>}
\text{ForwardIterator2}
\text{unique_copy(ExecutionPolicy&\& \text{exec},}
\text{ForwardIterator1 \text{first}, ForwardIterator1 \text{last},}
\text{ForwardIterator2 \text{result});}
\]
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<InputIterator I, Sentinel<I> S, WeaklyIncrementable O, class Proj = identity, IndirectRelation<projected<I, Proj>> C = ranges::equal_to>
requires IndirectlyCopyable<I, O> &&
(ForwardIterator<I> ||
(InputIterator<O> && Same<iter_value_t<I>, iter_value_t<O>>) ||
IndirectlyCopyableStorable<I, O>)
constexpr ranges::unique_copy_result<I, O>
ranges::unique_copy(I first, S last, O result, C comp = {}, Proj proj = {});

template<InputRange R, WeaklyIncrementable O, class Proj = identity, IndirectRelation<projected<iterator_t<R>, Proj>> C = ranges::equal_to>
requires IndirectlyCopyable<iterator_t<R>, O> &&
(ForwardIterator<iterator_t<R>> ||
(InputIterator<iterator_t<R> && Same<iter_value_t<iterator_t<R>>, iter_value_t<O>>) ||
IndirectlyCopyableStorable<iterator_t<R>, O>)
constexpr ranges::unique_copy_result<safe_iterator_t<R>, O>
ranges::unique_copy(R&& r, O result, C comp = {}, Proj proj = {});

6 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, or
   — bool(invoke(comp, invoke(proj, *i), invoke(proj, *(i - 1)))) for the overloads in
     namespace ranges.

7 Requires:
   — The ranges [first, last) and [result, result+(last-first)) shall not overlap.
   — For the overloads in namespace std:
     — The comparison function shall be an equivalence relation.
     — The expression *result = *first shall be valid.
     — For the overloads with no ExecutionPolicy, let T be the value type of InputIterator.
        If InputIterator meets the Cpp17ForwardIterator requirements, then there are no addi-
        tional requirements for T. Otherwise, if OutputIterator meets the Cpp17ForwardIterator
        requirements and its value type is the same as T, then T shall meet the Cpp17CopyAssign-
        able (Table 29) requirements. Otherwise, T shall meet both the Cpp17CopyConstructible
        (Table 27) and Cpp17CopyAssignable requirements. [Note: For the overloads with an Execu-
        tionPolicy, there may be a performance cost if the value type of ForwardIterator1 does
        not meet both the Cpp17CopyConstructible and Cpp17CopyAssignable requirements. — end
        note]

8 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.

9 Returns:
   — result + N for the overloads in namespace std, or
   — {last, result + N} for the overloads in namespace ranges

10 Complexity: Exactly last - first - 1 applications of the corresponding predicate and no more than
twice as many applications of any projection.

25.6.10 Reverse [alg.reverse]

template<class BidirectionalIterator>
constexpr void reverse(BidirectionalIterator first, BidirectionalIterator last);

```cpp
template<class ExecutionPolicy, class BidirectionalIterator>
void reverse(ExecutionPolicy&& exec,
             BidirectionalIterator first, BidirectionalIterator last);
```

```cpp
template<BidirectionalIterator I, Sentinel<I> S>
requires Permutable<I>
constexpr I ranges::reverse(I first, S last);
```

```cpp
template<BidirectionalRange R>
requires Permutable<iterator_t<R>>
constexpr safe_iterator_t<R> ranges::reverse(R&& r);
```

1. Requires: For the overloads in namespace std, BidirectionalIterator shall meet 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<BidirectionalIterator I, Sentinel<I> S, WeaklyIncrementable O>
requires IndirectlyCopyable<I, O>
constexpr ranges::reverse_copy_result<I, O>
ranges::reverse_copy(I first, S last, O result);
```

```cpp
template<BidirectionalRange R, WeaklyIncrementable O>
requires IndirectlyCopyable<iterator_t<R>, O>
constexpr ranges::reverse_copy_result<safe_iterator_t<R>, O>
ranges::reverse_copy(R&& r, O result);
```

5. Let N be last - first.
6. Requires: The ranges [first, last) and [result, result + N) shall 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:
   - (8.1) result + N for the overloads in namespace std, or
   - (8.2) {last, result + N} for the overloads in namespace ranges.

### 25.6.11 Rotate [alg.rotate]

```cpp
template<class ForwardIterator>
constexpr ForwardIterator
rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
rotate(ExecutionPolicy&& exec,
       ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```
template<Permutable I, Sentinel<I> S>  
constexpr subrange<I> ranges::rotate(I first, I middle, S last);  

Requires: [first, middle) and [middle, last) shall be valid ranges. For the overloads in namespace std, ForwardIterator shall meet the Cpp17ValueSwappable requirements (16.5.3.2), and the type of *first shall meet the Cpp17MoveConstructible (Table 26) and Cpp17MoveAssignable (Table 28) requirements.

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]

Returns:  
(3.1) first + (last - middle) for the overloads in namespace std, or  
(3.2) \{first + (last - middle), last\} for the overload in namespace ranges.

Complexity: At most last - first swaps.

template<ForwardRange R>  
requires Permutable<iterator_t<R>>  
constexpr safe_subrange_t<R> ranges::rotate(R&& r, iterator_t<R> middle);  

Effects: Equivalent to: return ranges::rotate(ranges::begin(r), middle, ranges::end(r));

template<class ForwardIterator, class OutputIterator>  
constexpr OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle, ForwardIterator last,  
OutputIterator result);  

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>  
ForwardIterator2 rotate_copy(ExecutionPolicy&& exec,  
ForwardIterator1 first, ForwardIterator1 middle, ForwardIterator1 last,  
ForwardIterator2 result);

template<ForwardIterator I, Sentinel<I> S, WeaklyIncrementable O>  
requires IndirectlyCopyable<I, O>  
constexpr ranges::rotate_copy_result<I, O>  
ranges::rotate_copy(I first, I middle, S last, O result);  

Let N be last - first.

Requires: [first, middle) and [middle, last) shall be valid ranges. The ranges [first, last) and [result, result + N) shall not overlap.

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).

Returns:  
(9.1) result + N for the overloads in namespace std, or  
(9.2) \{last, result + N\} for the overload in namespace ranges.

Complexity: Exactly N assignments.

template<ForwardRange R, WeaklyIncrementable O>  
requires IndirectlyCopyable<iterator_t<R>, O>  
constexpr ranges::rotate_copy_result<safe_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);

25.6.12 Sample  
[alg.random.sample]  

template<class PopulationIterator, class SampleIterator,  
class Distance, class UniformRandomBitGenerator>  
SampleIterator sample(PopulationIterator first, PopulationIterator last,
SampleIterator out, Distance n,
UniformRandomBitGenerator&& g);

Requires:
(1.1) — PopulationIterator shall satisfy the Cpp17InputIterator requirements (23.3.5.2).
(1.2) — SampleIterator shall satisfy the Cpp17OutputIterator requirements (23.3.5.3).
(1.3) — SampleIterator shall satisfy the Cpp17RandomAccessIterator requirements (23.3.5.6) unless PopulationIterator satisfies the Cpp17ForwardIterator requirements (23.3.5.4).
(1.4) — PopulationIterator’s value type shall be writable (23.3.1) to out.
(1.5) — Distance shall be an integer type.
(1.6) — remove_reference_t<UniformRandomBitGenerator> shall satisfy the requirements of a uniform random bit generator type (26.6.2.3) whose return type is convertible to Distance.
(1.7) — out shall not be in the range [first, last).

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: \(\Theta(last - first)\).

Remarks:
(5.1) — Stable if and only if PopulationIterator satisfies the Cpp17ForwardIterator requirements.
(5.2) — To the extent that the implementation of this function makes use of random numbers, the object g shall serve as the implementation’s source of randomness.

25.6.13 Shuffle

template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first,
RandomAccessIterator last,
UniformRandomBitGenerator&& g);

template<RandomAccessIterator I, Sentinel<I> S, class Gen>
requires Permutable<I> &&
UniformRandomBitGenerator<remove_reference_t<Gen>>
I ranges::shuffle(I first, S last, Gen&& g);

template<RandomAccessRange R, class Gen>
requires Permutable<iterator_t<R>> &&
UniformRandomBitGenerator<remove_reference_t<Gen>>
safe_iterator_t<R> ranges::shuffle(R&& r, Gen&& g);

Requires: For the overload in namespace std:
(1.1) — RandomAccessIterator shall meet the Cpp17ValueSwappable requirements (16.5.3.2).
(1.2) — The type remove_reference_t<UniformRandomBitGenerator> shall meet 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.

25.6.14 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);

1 Requires: The type of *first shall satisfy the Cpp17MoveAssignable requirements.

2 Effects: If n \leq 0 or n \geq last - first, does nothing. Otherwise, moves the element from position first + n + i into position first + i for each non-negative integer i < (last - first) - n. In the first overload case, does so in order starting from i = 0 and proceeding to i = (last - first) - n - 1.

3 Returns: first + (last - first - n) if n is positive and n < last - first, otherwise first if n is positive, otherwise last.

4 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);

5 Requires: The type of *first shall satisfy the Cpp17MoveAssignable requirements. ForwardIterator shall meet the Cpp17BidirectionalIterator requirements (23.3.5.5) or the Cpp17ValueSwappable requirements.

6 Effects: If n \leq 0 or n \geq 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 satisfies the Cpp17BidirectionalIterator requirements, does so in order starting from i = (last - first) - n - 1 and proceeding to i = 0.

7 Returns: first + n if n is positive and n < last - first, otherwise last if n is positive, otherwise first.

8 Complexity: At most (last - first) - n assignments or swaps.

25.7 Sorting and related operations

The operations in 25.7 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 25.7.3, 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:

- comp(a, b) && comp(b, c) implies comp(a, c)
- equiv(a, b) && equiv(b, c) implies equiv(a, c)

[Note: Under these conditions, it can be shown that

- equiv is an equivalence relation,
- comp induces a well-defined relation on the equivalence classes determined by equiv, and
- the induced relation is a strict total ordering.

-end note]
A sequence is sorted with respect to a \texttt{comp} and \texttt{proj} for a comparator and projection \texttt{comp} and \texttt{proj} if for every iterator \(i\) pointing to the sequence and every non-negative integer \(n\) such that \(i + n\) is a valid iterator pointing to an element of the sequence,

\[
\text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, \star(i + n)), \text{invoke}(\text{proj}, \star i)))
\]

is false.

A sequence \([\text{start}, \text{finish})\) is partitioned with respect to an expression \(f(e)\) if there exists an integer \(n\) such that for all \(0 \leq i < (\text{finish} - \text{start})\), \(f(\text{*(start} + i))\) is true if and only if \(i < n\).

In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an \texttt{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 \(!((a < b) \&\& (!b < a))\).

### 25.7.1 Sorting

#### 25.7.1.1 \texttt{sort}

```cpp
template<class RandomAccessIterator>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator>
void sort(ExecutionPolicy&& exec,
          RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr void sort(RandomAccessIterator first, RandomAccessIterator last,
                    Compare comp);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(ExecutionPolicy&& exec,
          RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

```cpp
template<RandomAccessIterator I, Sentinel<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 = {});
```

Let \texttt{comp} be \texttt{less{}} and \texttt{proj} be \texttt{identity{}} for the overloads with no parameters by those names.

**Requires:** For the overloads in namespace \texttt{std}, \texttt{RandomAccessIterator} shall meet the \texttt{Cpp17ValueSwappable} requirements (16.5.3.2) and the type of \texttt{*first} shall meet the \texttt{Cpp17MoveConstructible} (Table 26) and \texttt{Cpp17MoveAssignable} (Table 28) requirements.

**Effects:** Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \texttt{comp} and \texttt{proj}.

**Returns:** \texttt{last}, for the overloads in namespace \texttt{ranges}.

**Complexity:** Let \(N\) be \texttt{last} - \texttt{first}. \(\Theta(N \log N)\) comparisons and projections.

#### 25.7.1.2 \texttt{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);
```

§ 25.7.1.2
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  void stable_sort(ExecutionPolicy&& exec,
                  RandomAccessIterator first, RandomAccessIterator last,
                  Compare comp);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
  I ranges::stable_sort(I first, S last, Comp comp = {}, Proj proj = {});

template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
  safe_iterator_t<R> ranges::stable_sort(R&& r, Comp comp = {}, Proj proj = {});

1. Let \( \text{comp} \) be \( \text{less}{} \) and \( \text{proj} \) be \( \text{identity}{} \) for the overloads with no parameters by those names.
2. Requires: For the overloads in namespace \( \text{std} \), \( \text{RandomAccessIterator} \) shall meet the \( \text{Cpp17ValueSwappable} \) requirements (16.5.3.2) and the type of \( \ast \text{first} \) shall meet the \( \text{Cpp17MoveConstructible} \) (Table 26) and \( \text{Cpp17MoveAssignable} \) (Table 28) requirements.
3. Effects: Sorts the elements in the range \([\text{first}, \text{last})\) with respect to \( \text{comp} \) and \( \text{proj} \).
4. Returns: \( \text{last} \) for the overloads in namespace \( \text{ranges} \).
5. 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.
6. Remarks: Stable (16.5.5.7).

25.7.1.3 partial_sort

\[
\text{template<RandomAccessIterator>}
\begin{align*}
  &\text{constexpr void partial_sort(RandomAccessIterator first,} \\
  &\text{ RandomAccessIterator middle,} \\
  &\text{ RandomAccessIterator last);}
\end{align*}
\]

\[
\text{template<ExecutionPolicy, class RandomAccessIterator>}
\begin{align*}
  &\text{void partial_sort(ExecutionPolicy&& exec,} \\
  &\text{ RandomAccessIterator first,} \\
  &\text{ RandomAccessIterator middle,} \\
  &\text{ RandomAccessIterator last);}
\end{align*}
\]

\[
\text{template<RandomAccessIterator, class Compare>}
\begin{align*}
  &\text{constexpr void partial_sort(RandomAccessIterator first,} \\
  &\text{ RandomAccessIterator middle,} \\
  &\text{ RandomAccessIterator last,} \\
  &\text{ Compare comp);}
\end{align*}
\]

\[
\text{template<ExecutionPolicy, class RandomAccessIterator, class Compare>}
\begin{align*}
  &\text{void partial_sort(ExecutionPolicy&& exec,} \\
  &\text{ RandomAccessIterator first,} \\
  &\text{ RandomAccessIterator middle,} \\
  &\text{ RandomAccessIterator last,} \\
  &\text{ Compare comp);}
\end{align*}
\]

\[
\text{template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,}
\text{ class Proj = identity>}
\begin{align*}
  &\text{I ranges::partial_sort(I first, I middle, S last, Comp comp = {}, Proj proj = {});}
\end{align*}
\]

1. Let \( \text{comp} \) be \( \text{less}{} \) and \( \text{proj} \) be \( \text{identity}{} \) for the overloads with no parameters by those names.
2. Requires: \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) shall be valid ranges. For the overloads in namespace \( \text{std} \), \( \text{RandomAccessIterator} \) shall meet the \( \text{Cpp17ValueSwappable} \) requirements (16.5.3.2) and the type of \( \ast \text{first} \) shall meet the \( \text{Cpp17MoveConstructible} \) (Table 26) and \( \text{Cpp17MoveAssignable} \) (Table 28) requirements.

§ 25.7.1.3
Effects: Places the first \(\text{middle} - \text{first}\) elements from the range \([\text{first}, \text{last})\) as sorted with respect to \(\text{comp}\) and \(\text{proj}\) into the range \([\text{first}, \text{middle})\). The rest of the elements in the range \([\text{middle}, \text{last})\) are placed in an unspecified order.

Returns: \(\text{last}\) for the overload in namespace \text{ranges}.

Complexity: Approximately \((\text{last} - \text{first}) \times \log(\text{middle} - \text{first})\) comparisons, and twice as many projections.

g1

\[
\begin{align*}
\text{template}<\text{RandomAccessRange} \ R, \ \text{class} \ \text{Comp} = \text{ranges}::\text{less}, \ \text{class} \ \text{Proj} = \text{identity}>
& \quad \text{requires Sortable<iterator_t<\text{R}>, \ Comp, \ Proj>}
& \quad \text{constexpr safe_iterator_t<\text{R}>}
& \quad \text{ranges::partial_sort(\&\& \text{r}, \ \text{iterator_t<\text{R}> middle, \ Comp \ comp = {}}, \ \text{Proj proj = {}})};
\end{align*}
\]

\text{Effects: Equivalent to:}

\[
\quad \text{return ranges::partial_sort(\text{ranges}::\text{begin(r)}, \ \text{middle}, \ \text{ranges}::\text{end(r)}, \ \text{comp}, \ \text{proj})};
\]

\text{25.7.1.4 partial_sort_copy}

\[
\begin{align*}
\text{template}<\text{class InputIterator}, \ \text{class RandomAccessIterator}>
& \quad \text{constexpr RandomAccessIterator}
& \quad \text{partial_sort_copy(InputIterator first, \ InputIterator last,}
& \quad \text{RandomAccessIterator result_first,}
& \quad \text{RandomAccessIterator result_last);}\\
\text{template}<\text{class ExecutionPolicy}, \ \text{class ForwardIterator}, \ \text{class RandomAccessIterator}>
& \quad \text{RandomAccessIterator}
& \quad \text{partial_sort_copy(ExecutionPolicy&& \ exec,}
& \quad \text{ForwardIterator first, \ ForwardIterator last,}
& \quad \text{RandomAccessIterator result_first,}
& \quad \text{RandomAccessIterator result_last);}\\
\text{template}<\text{class InputIterator}, \ \text{class RandomAccessIterator},
& \quad \text{class Compare}>
& \quad \text{constexpr RandomAccessIterator}
& \quad \text{partial_sort_copy(InputIterator first, \ InputIterator last,}
& \quad \text{RandomAccessIterator result_first,}
& \quad \text{RandomAccessIterator result_last,}
& \quad \text{Compare \ comp);}\\
\text{template}<\text{class ExecutionPolicy}, \ \text{class ForwardIterator}, \ \text{class RandomAccessIterator},
& \quad \text{class Compare}>
& \quad \text{RandomAccessIterator}
& \quad \text{partial_sort_copy(ExecutionPolicy&& \ exec,}
& \quad \text{ForwardIterator first, \ ForwardIterator last,}
& \quad \text{RandomAccessIterator result_first,}
& \quad \text{RandomAccessIterator result_last,}
& \quad \text{Compare \ comp);}\\
\text{template}<\text{InputIterator I1, \ Sentinel<I1> S1, \ RandomAccessIterator I2, \ Sentinel<I2> S2,}
& \quad \text{class Compare = \text{ranges}::\text{less}, \ \text{class} \ \text{Proj1 = identity, \ class} \ \text{Proj2 = identity}>
& \quad \text{requires IndirectlyCopyable<I1, \ I2> \ & IndirectlyCopyable<I2, \ Comp, \ Proj2> \ & IndirectlyCopyable<I1, \ Proj1>, \ projected<I2, \ Proj2>>}
& \quad \text{constexpr I2}
& \quad \text{ranges::partial_sort_copy(I1 first, \ S1 last, \ I2 result_first, \ S2 result_last,}
& \quad \text{Comp comp = {}}, \ \text{Proj1 proj1 = {}}, \ \text{Proj2 proj2 = {}});}\\
\text{template}<\text{InputRange R1, \ RandomAccessRange R2, \ class Compare = \text{ranges}::\text{less},}
& \quad \text{class Proj1 = identity, \ class Proj2 = identity}>
& \quad \text{requires IndirectlyCopyable<iterator_t<R1>, \ iterator_t<R2>> \ & Sortable<iterator_t<R2>, \ Comp, \ Proj2> \ & IndirectlyCopyable<Comp, \ projected<iterator_t<R1>, \ Proj1>, \ projected<iterator_t<R2>, \ Proj2>>}
& \quad \text{constexpr safe_iterator_t<\text{R2}>}
& \quad \text{ranges::partial_sort_copy(R1\&\& \ r, \ R2\&\& \ result_r, \ Comp \ comp = {}},
& \quad \text{Proj1 proj1 = {}}, \ \text{Proj2 proj2 = {}});}\\
\end{align*}
\]

Let \(N\) be \(\min(\text{last} - \text{first}, \ \text{result_last} - \text{result_first})\). Let \(\text{comp}\) be \text{less}{}\), and \(\text{proj1}\) and \(\text{proj2}\) be \text{identity}{} for the overloads with no parameters by those names.
Requires: For the overloads in namespace std, RandomAccessIterator shall meet the Cpp17Value-Swappable requirements (16.5.3.2), the type of *result_first shall meet the Cpp17MoveConstructible (Table 26) and Cpp17MoveAssignable (Table 28) requirements, and the expression *first shall be writable (23.3.1) to result_first.

Expects: 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

\[
\text{bool}(\text{invoke(comp, invoke(proj1, *a1), invoke(proj2, *y2)}). \]

Then, after evaluating the assignment *x2 = *a1, E is equal to

\[
\text{bool}(\text{invoke(comp, invoke(proj2, *x2), invoke(proj2, *y2)}). \]

[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 \([\text{result}_-\text{first}, \text{result}_\text{first} + \text{N})\).

Returns: result_first + N.

Complexity: Approximately \((last - first) \times \log N\) comparisons, and twice as many projections.

25.7.1.5 \texttt{is\_sorted}

\[
\text{template}<\text{class ForwardIterator}> \\
\text{constexpr bool is\_sorted(ForwardIterator first, ForwardIterator last);} \]

Effects: Equivalent to:

\[
\text{return is\_sorted\_until(first, last) == last;} \]

\[
\text{template}<\text{class ExecutionPolicy, class ForwardIterator}> \\
\text{bool is\_sorted(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last);} \]

Effects: Equivalent to:

\[
\text{return is\_sorted\_until(std::forward<ExecutionPolicy>(exec), first, last) == last;} \]

\[
\text{template}<\text{class ForwardIterator, class Compare}> \\
\text{constexpr bool is\_sorted(ForwardIterator first, ForwardIterator last,} \\
\text{Compare comp);} \]

Effects: Equivalent to:

\[
\text{return is\_sorted\_until(first, last, comp) == last;} \]

\[
\text{template}<\text{class ExecutionPolicy, class ForwardIterator, class Compare}> \\
\text{bool is\_sorted(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last,} \\
\text{Compare comp);} \]

Effects: Equivalent to:

\[
\text{return is\_sorted\_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;} \]

\[
\text{template<ForwardIterator I, Sentinel<I> S, class Proj = identity,} \\
\text{IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>} \\
\text{constexpr bool ranges::is\_sorted(I first, S last, Comp comp = {}, Proj proj = {});} \]

\[
\text{template<ForwardRange R, class Proj = identity,} \\
\text{IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less>} \\
\text{constexpr bool ranges::is\_sorted(R&& r, Comp comp = {}, Proj proj = {});} \]

Effects: Equivalent to:

\[
\text{return ranges::is\_sorted\_until(first, last, comp, proj) == last;} \]

\[
\text{template<\text{class ForwardIterator}>} \\
\text{constexpr ForwardIterator} \\
\text{is\_sorted\_until(ForwardIterator first, ForwardIterator last);} \]

\[
\text{template<\text{class ExecutionPolicy, class ForwardIterator}>} \\
\text{ForwardIterator} \\
\text{is\_sorted\_until(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last);} \]
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<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
constexpr I
ranges::is_sorted_until(I first, S last, Comp comp = {}, Proj proj = {});

template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<iterator_t<R>, Proj>>
constexpr safe_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.

25.7.2 Nth element
[alg.nth.element]

template<class RandomAccessIterator>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                           RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator>
void nth_element(ExecutionPolicy&& exec,
                 RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                           RandomAccessIterator last, Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void nth_element(ExecutionPolicy&& exec,
                 RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
         class Proj = identity>
requires Sortable<I, S, Comp = ranges::less, Proj = identity>
constexpr I
ranges::nth_element(I first, I nth, S last, Comp comp = {}, Proj proj = {});

Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.

Requires: (first, nth) and (nth, last) shall be valid ranges. For the overloads in namespace std, RandomAccessIterator shall meet the Cpp17ValueSwappable requirements (16.5.3.2), and the type of *first shall meet the Cpp17MoveConstructible (Table 26) and Cpp17MoveAssignable (Table 28) requirements.

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.

Returns: last for the overload in namespace ranges.

Complexity: For the overloads with no ExecutionPolicy, linear on average. For the overloads with an ExecutionPolicy, \( O(N) \) applications of the predicate, and \( O(N \log N) \) swaps, where \( N = \text{last} - \text{first} \).
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R>
ranges::nth_element(R&& r, iterator_t<R> nth, Comp comp = {}, Proj proj = {});

Effects: Equivalent to:

```
return ranges::nth_element(ranges::begin(r), nth, ranges::end(r), comp, proj);
```

### 25.7.3 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.

#### 25.7.3.1 lower_bound

```cpp
template<class ForwardIterator, class T>
constexpr ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
const T& value);
```

```cpp
template<class ForwardIterator, class T, class Compare>
constexpr ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

```cpp
template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::lower_bound(I first, S last, const T& value, Comp comp = {}, Proj proj = {});
```

```cpp
template<ForwardRange R, class T, class Proj = identity,
IndirectStrictWeakOrder<iterator_t<R>, Proj>> Comp = ranges::less>
constexpr safe_iterator_t<R>
ranges::lower_bound(R&& r, const T& value, Comp comp = {}, Proj proj = {});
```

1. Let \( \text{comp} \) be \text{less{}} and \( \text{proj} \) be \text{identity{}} for overloads with no parameters by those names.
2. Requires: The elements \( e \) of \([\text{first}, \text{last})\) shall be partitioned with respect to the expression

\[
\text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, e), value)).
\]

3. Returns: The furthermost iterator \( i \) in the range \([\text{first}, \text{last})\) such that for every iterator \( j \) in the range \([\text{first}, j)\), 
\[
\text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj}, *j), value)) \text{ is true.}
\]

4. Complexity: At most \( \log_2(last - first) + \Theta(1) \) comparisons and projections.

#### 25.7.3.2 upper_bound

```cpp
template<class ForwardIterator, class T>
constexpr ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
const T& value);
```

```cpp
template<class ForwardIterator, class T, class Compare>
constexpr ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
const T& value, Compare comp);
```

```cpp
template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::upper_bound(I first, S last, const T& value, Comp comp = {}, Proj proj = {});
```

§ 25.7.3.2
template<ForwardRange R, class T, class Proj = identity,  
    IndirectStrictWeakOrder<const T*, projected<iterator_t<R>, Proj>> Comp =  
    ranges::less>
constexpr safe_iterator_t<R>
    ranges::upper_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.

Requires: The elements e of [first, last) shall be partitioned with respect to the expression
    !bool(invoke(comp, value, invoke(proj, e))).

Returns: The furthestmost 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 \(\log_2(last - first) + \Theta(1)\) comparisons and projections.

25.7.3.3 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<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,  
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = ranges::less>
constexpr subrange<I>
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.

Requires: The elements e of [first, last) shall be 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)) shall imply !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 \(2 * \log_2(last - first) + \Theta(1)\) comparisons and projections.

25.7.3.4 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<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
         IndirectStrictWeakOrder<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<ForwardRange R, class T, class Proj = identity,
         IndirectStrictWeakOrder<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 \( \text{comp} \) be \text{less}{} and \( \text{proj} \) be \text{identity}{} for overloads with no parameters by those names.

\section{25.7.4 Partitions [alg.partitions]}

\begin{verbatim}
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);
\end{verbatim}

\begin{verbatim}
template<InputIterator I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred = identity>
constexpr bool ranges::is_partitioned(I first, S last, Pred pred, Proj proj = {});
template<InputRange R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred = identity>
constexpr bool ranges::is_partitioned(R&& r, Pred pred, Proj proj = {});
\end{verbatim}

Let \( \text{proj} \) be \text{identity}{} for the overloads with no parameter named \( \text{proj} \).

\begin{verbatim}
template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
partition(ForwardIterator first, ForwardIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator
partition(ExecutionPolicy&& exec,
          ForwardIterator first, ForwardIterator last, Predicate pred);
\end{verbatim}

\begin{verbatim}
template<Permutable I, Sentinel<I> S, class Proj = identity,
         IndirectUnaryPredicate<projected<I, Proj>> Pred = identity>
constexpr I
ranges::partition(I first, S last, Pred pred, Proj proj = {});
template<Permutable R, class Proj = identity,
         IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred = identity>
requires Permutable<iterator_t<R>>
constexpr safe_iterator_t<R>
ranges::partition(R&& r, Pred pred, Proj proj = {});
\end{verbatim}

Let \( \text{proj} \) be \text{identity}{} for the overloads with no parameter named \( \text{proj} \) and let \( E(x) \) be \text{bool}(\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, x))) \).

\begin{verbatim}
Requires: For the overloads in namespace std, \text{ForwardIterator} shall meet the Cpp17ValueSwappable
requirements (16.5.3.2).
\end{verbatim}
Effects: Places all the elements e in [first, last) that satisfy E(e) before all the elements that do not.

Returns: An iterator i such that E(*j) is true for every iterator j in [first, i) and false for every iterator j in [i, last).

Complexity: Let N = last - first:

(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 BidirectionalIterator for the overloads in namespace ranges, and at most N swaps otherwise.

(8.2) For the overload with an ExecutionPolicy, $O(N \log N)$ swaps and $O(N)$ applications of the predicate.

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,
BidirectionalIterator first, BidirectionalIterator last, Predicate pred);

template<BidirectionalIterator I, Sentinel<I> S, class Proj = identity,
IndirectUnaryPredicate<projected<I, Proj>> Pred>
requires Permutable<I>
I ranges::stable_partition(I first, S last, Pred pred, Proj proj = {});

template<BidirectionalRange R, class Proj = identity,
IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
requires Permutable<iterator_t<R>>
safe_iterator_t<R> ranges::stable_partition(R&& r, Pred pred, Proj proj = {});

Let proj be identity{} for the overloads with no parameter named proj and let E(x) be bool(invoke(pred, invoke(proj, x))).

Requires: For the overloads in namespace std, BidirectionalIterator shall meet the Cpp17Value-Swappable requirements (16.5.3.2) and the type of *first shall meet the Cpp17MoveConstructible (Table 26) and Cpp17MoveAssignable (Table 28) 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: An iterator i 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,

Complexity: Let N = last - first:

(12.1) 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.

(12.2) For the overload with an ExecutionPolicy, $O(N \log N)$ swaps and $O(N)$ applications of the predicate.

template<class InputIterator, class OutputIterator1,
class OutputIterator2, class Predicate>
constexpr pair<OutputIterator1, OutputIterator2>
partition_copy(InputIterator first, InputIterator last,
OutputIterator1 out_true, OutputIterator2 out_false, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class ForwardIterator1,
class ForwardIterator2, class Predicate>
pair<ForwardIterator1, ForwardIterator2>
partition_copy(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last,
ForwardIterator1 out_true, ForwardIterator2 out_false, Predicate pred);
```cpp
template<InputIterator I, Sentinel<I> S, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity, IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, O1> && IndirectlyCopyable<I, O2>
    constexpr ranges::partition_copy_result<I, O1, O2>
    ranges::partition_copy(I first, S last, O1 out_true, O2 out_false, Pred pred, 
    Proj proj = {});

template<InputRange R, WeaklyIncrementable O1, WeaklyIncrementable O2,
    class Proj = identity,
    IndirectUnaryPredicate<iterator_t<R>, Proj> Pred>
    requires IndirectlyCopyable<iterator_t<R>, O1> &&
    IndirectlyCopyable<iterator_t<R>, O2>
    constexpr ranges::partition_copy_result<safe_iterator_t<R>, O1, O2>
    ranges::partition_copy(R&& r, O1 out_true, O2 out_false, Pred pred, Proj proj = {});
```

Let `proj` be `identity{}` for the overloads with no parameter named `proj` and let `E(x)` be `bool(invoke(pred, invoke(proj, x)))`.

**Requires:** The input range and output ranges shall not overlap. For the overloads in namespace `std`, the expression `*first` shall be writable (23.3.1) to `out_true` and `out_false`. [Note: For the overload with an `ExecutionPolicy`, there may be a performance cost if `first`'s value type does not meet the `Cpp17CopyConstructible` requirements. — end note]

**Effects:** For each iterator `i` in `[first, last)`, copies `*i` to the output range beginning with `out_true` if `E(*i)` is `true`, or to the output range beginning with `out_false` otherwise.

**Returns:** Let `o1` be the end of the output range beginning at `out_true`, and `o2` the end of the output range beginning at `out_false`. Returns

- `{o1, o2}` for the overloads in namespace `std`, or
- `{last, o1, o2}` for the overloads in namespace `ranges`.

**Complexity:** Exactly `last - first` applications of `pred` and `proj`.

```cpp
template<class ForwardIterator, class Predicate>
    constexpr ForwardIterator
    partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
    IndirectUnaryPredicate<projected<I, Proj>> Pred>
    requires IndirectlyCopyable<I, Proj>> Pred>
    constexpr I ranges::partition_point(I first, S last, Pred pred, Proj proj = {});

template<ForwardRange R, class Proj = identity,
    IndirectUnaryPredicate<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
    ranges::partition_point(R&& r, Pred pred, Proj proj = {});
```

Let `proj` be `identity{}` for the overloads with no parameter named `proj` and let `E(x)` be `bool(invoke(pred, invoke(proj, x)))`.

**Requires:** The elements `e` of `[first, last)` shall be partitioned with respect to `E(e).

**Returns:** An iterator `mid` such that `E(*i)` is `true` for all iterators `i` in `[first, mid)`, and `false` for all iterators `i` in `[mid, last)`

**Complexity:** `Θ(log(last - first))` applications of `pred` and `proj`.

### 25.7.5 Merge [alg.merge]

```cpp
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);
```
```

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 Compare>
    ForwardIterator
    merge(ExecutionPolicy&& exec,
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        ForwardIterator result, Compare comp);

template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable 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<InputRange R1, InputRange R2, WeaklyIncrementable 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::merge_result<safe_iterator_t<R1>, safe_iterator_t<R2>, O>
    ranges::merge(R1&& r1, R2&& r2, O result,
        Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

Let \( N \) be \((last1 - first1) + (last2 - first2)\). Let \( \text{comp} \) be \(\text{less}\), \(\text{proj1} \) be \(\text{identity}\), and \(\text{proj2} \) be \(\text{identity}\), for the overloads with no parameters by those names.

Requires: The ranges \([first1, last1)\) and \([first2, last2)\) shall be sorted with respect to \(\text{comp}\) and \(\text{proj1}\) or \(\text{proj2}\), respectively. The resulting range shall 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 \(\text{bool}(\text{invoke}(\text{comp}, \text{invoke}(\text{proj2}, e2), \text{invoke}(\text{proj1}, e1)))\) is true.

Returns:
1. \(result\_last\) for the overloads in namespace std, or
2. \{last1, last2, result\_last\} for the overloads in namespace ranges.

Complexity:
1. For the overloads with no ExecutionPolicy, at most \(N - 1\) comparisons and applications of each projection.
2. For the overloads with an ExecutionPolicy, \(O(N)\) comparisons.

Remarks: Stable (16.5.5.7).
```

\[ 25.7.5 \]
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);

template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
    class Proj = identity>
requires Sortable<I, Comp, Proj>
I ranges::inplace_merge(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.

Requires: [first, middle) and [middle, last) shall be valid ranges sorted with respect to comp
and proj. For the overloads in namespace std, BidirectionalIterator shall meet the Cpp17Value-
Swapable requirements (16.5.3.2) and the type of *first shall meet the Cpp17MoveConstructible
(Table 26) and Cpp17MoveAssignable (Table 28) 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 comp
and proj.

Returns: last for the overload in namespace ranges.

Complexity: Let N = last - first:

- For the overloads with no ExecutionPolicy, and if enough additional memory is available, exactly
  N - 1 comparisons.
- Otherwise, \( O(N \log N) \) comparisons.

In either case, twice as many projections as comparisons.

Remarks: Stable (16.5.5.7).

template<BidirectionalRange R, class Comp = ranges::less, class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
safe_iterator_t<R>
ranges::inplace_merge(R&& r, iterator_t<R> middle, Comp comp = {}, Proj proj = {});

Effects: Equivalent to:

return ranges::inplace_merge(ranges::begin(r), middle, ranges::end(r), comp, proj);

25.7.6 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.

25.7.6.1 includes

```cpp
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<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    class Proj1 = identity, class Proj2 = identity,
    IndirectStrictWeakOrder<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<InputRange R1, InputRange R2, class Proj1 = identity,
    class Proj2 = identity,
    IndirectStrictWeakOrder<projected<iterator_t<R1>, Proj1>,
        projected<iterator_t<R2>, Proj2>> Comp = ranges::less>
constexpr bool ranges::includes(R1&& r1, R2&& r2, Comp comp = {},
    Proj1 proj1 = {}, Proj2 proj2 = {});

Let comp be less{}, proj1 be identity{}, and proj2 be identity{}, for the overloads with no
parameters by those names.

Requires: The ranges [first1, last1) and [first2, last2) shall be sorted with respect to comp
and proj1 or proj2, respectively.

Returns: true if and only if [first2, last2) is a subsequence of [first1, last1). [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 * ((last1 - first1) + (last2 - first2)) - 1 comparisons and applica-
tions of each projection.

25.7.6.2 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<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
    WeaklyIncrementable O, class Comp = ranges::less,
    class Proj1 = identity, class Proj2 = identity>
requires Mergeable<I1, I2, O, Comp, Proj1, Proj2>
Let \( \text{comp} \) be \texttt{less{}}{}, and \( \text{proj}_1 \) and \( \text{proj}_2 \) be \texttt{identity{}}{} for the overloads with no parameters by those names.

2. **Requires:** The ranges \([\text{first}_1, \text{last}_1)\) and \([\text{first}_2, \text{last}_2)\) shall be sorted with respect to \( \text{comp} \) and \( \text{proj}_1 \) or \( \text{proj}_2 \), respectively. The resulting range shall not overlap with either of the original ranges.

3. **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.

4. **Returns:** 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

\[ \text{result}_\text{last} \] for the overloads in namespace \texttt{std}, or
\[ \{\text{last}_1, \text{last}_2, \text{result}_\text{last}\} \] for the overloads in namespace \texttt{ranges}.

5. **Complexity:** At most \( 2 \times ((\text{last}_1 - \text{first}_1) + (\text{last}_2 - \text{first}_2)) - 1 \) comparisons and applications of each projection.

6. **Remarks:** Stable (16.5.5.7). 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.
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, WeaklyIncrementable 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, O result, Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

template<InputRange R1, InputRange R2, WeaklyIncrementable 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<safe_iterator_t<R1>, safe_iterator_t<R2>, O> ranges::set_intersection(R1&& r1, R2&& r2, O result, Comp comp = {}, Proj1 proj1 = {}, Proj2 proj2 = {});

1 Let \( \text{comp} \) be \( \text{less}() \), and \( \text{proj1} \) and \( \text{proj2} \) be \( \text{identity}() \) for the overloads with no parameters by those names.

2 Requires: The ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) shall be sorted with respect to \( \text{comp} \) and \( \text{proj1} \) or \( \text{proj2} \), respectively. The resulting range shall not overlap with either of the original ranges.

3 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.

4 Returns: Let \( \text{result}_\text{last} \) be the end of the constructed range. Returns

\( \text{(4.1)} \) Let \( \text{result}_\text{last} \) be the end of the constructed range. Returns

\( \text{(4.2)} \) Let \( \text{result}_\text{last} \) be the end of the constructed range. Returns

5 Complexity: At most \( 2 * ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1 \) comparisons and applications of each projection.

6 Remarks: Stable (16.5.5.7). 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.

25.7.6.4 set_difference

[set.difference]
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2, WeaklyIncrementable 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 = {});

template<InputRange R1, InputRange R2, WeaklyIncrementable 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<safe_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.

Requires: The ranges \([\text{first1}, \text{last1})\) and \([\text{first2}, \text{last2})\) shall be sorted with respect to \( \text{comp} \) and \( \text{proj1} \) or \( \text{proj2} \), respectively. The resulting range shall 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 \( \text{result}\_\text{last} \) be the end of the constructed range. Returns

- (4.1) \( \text{result}\_\text{last} \) for the overloads in namespace \( \text{std} \), or
- (4.2) \{\text{last1}, \text{result}\_\text{last}\} for the overloads in namespace \( \text{ranges} \).

Complexity: At most \( 2 \ast ((\text{last1}\_\text{-}\text{first1}) + (\text{last2}\_\text{-}\text{first2})) - 1 \) 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.

# 25.7.6.5 set_symmetric_difference

\([\text{set.symmetric.difference}]\)

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 Compare>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec, ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, ForwardIterator result, Compare comp);
template<InputIterator I1, Sentinel<I1> S1, InputIterator I2, Sentinel<I2> S2,
WeaklyIncrementable 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<InputRange R1, InputRange R2, WeaklyIncrementable 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<safe_iterator_t<R1>, safe_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.

Requires: The resulting range shall not overlap with either of the original ranges. The ranges [first1,
last1) and [first2, last2) shall be sorted with respect to comp and proj1 or proj2, respectively.
The resulting range shall not overlap with either of the original ranges.

Effects: Copies the elements of the range [first1, last1) that are not present in the range [first2,
last2), and the elements of the range [first2, last2) that are not present in the range [first1,
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

- result_last for the overloads in namespace std, or
- {last1, last2, result_last} for the overloads in namespace ranges.

Complexity: At most $2 \times ((last1 - first1) + (last2 - first2)) - 1$ comparisons and applica-
tions of each projection.

Remarks: Stable (16.5.5.7). 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.

25.7.7 Heap operations

A random access range $[a, b)$ is a heap with respect to comp and proj for a comparator and projection
comp and proj if its elements are organized such that:

- 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.
- $*a$ may be removed by pop_heap, or a new element added by push_heap, in $O(\log N)$ time.

These properties make heaps useful as priority queues.

make_heap converts a range into a heap and sort_heap turns a heap into a sorted sequence.

25.7.7.1 push_heap

template<class RandomAccessIterator>
constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr void push_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<RandomAccessIterator I, Sentinel<I> S, class Comp = ranges::less,
class Proj = identity>
requires Sortable<I, Comp, Proj>
constexpr I
ranges::push_heap(I first, S last, Comp comp = {}, Proj proj = {});
Let \( \text{comp} \) be \text{less\{\}} and \( \text{proj} \) be \text{identity\{\}} for the overloads with no parameters by those names.

**Requires:** The range \([\text{first}, \text{last} - 1)\) shall be a valid heap with respect to \( \text{comp} \) and \( \text{proj} \). For the overloads in namespace \text{std}, the type of \(*\text{first} \) shall meet the \text{Cpp17MoveConstructible} requirements (Table 26) and the \text{Cpp17MoveAssignable} requirements (Table 28).

**Effects:** Places the value in the location \( \text{last} - 1 \) into the resulting heap \([\text{first}, \text{last})\).

**Returns:** \( \text{last} \) for the overloads in namespace \text{ranges}.

**Complexity:** At most \( \log(\text{last} - \text{first}) \) comparisons and twice as many projections.

### 25.7.7.3 make_heap

[make.heap]

```cpp
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<RandomAccessIterator I, Sentinel<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 = {});

Let \( \text{comp} \) be \text{less\{\}} and \( \text{proj} \) be \text{identity\{\}} for the overloads with no parameters by those names.
```

§ 25.7.7.3 1072
Requires: For the overloads in namespace `std`, the type of `*first` shall meet the `Cpp17MoveConstructible` (Table 26) and `Cpp17MoveAssignable` (Table 28) 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.

### 25.7.7.4 `sort_heap` [sort.heap]

```cpp
template<class RandomAccessIterator>
constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

```cpp
template<RandomAccessIterator I, Sentinel<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 = {});
```

```cpp
template<RandomAccessRange R, class Comp = ranges::less, class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R>
ranges::sort_heap(R&& r, Comp comp = {}, Proj proj = {});
```

1. Let `comp` be `less{}` and `proj` be `identity{}` for the overloads with no parameters by those names.

2. Requires: The range `[first, last)` shall be a valid heap with respect to `comp` and `proj`. For the overloads in namespace `std`, `RandomAccessIterator` shall meet the `Cpp17ValueSwappable` requirements (16.5.3.2) and the type of `*first` shall meet the `Cpp17MoveConstructible` (Table 26) and `Cpp17MoveAssignable` (Table 28) requirements.

3. Effects: Sorts elements in the heap `[first, last)` with respect to `comp` and `proj`.

4. Returns: `last` for the overloads in namespace `ranges`.

5. Complexity: At most 2N log N comparisons, where N = last - first, and twice as many projections.

### 25.7.7.5 `is_heap` [is.heap]

```cpp
template<class RandomAccessIterator>
constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator>
bool is_heap(ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last);
```

```cpp
template<class RandomAccessIterator, class Compare>
constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

```cpp
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
bool is_heap(ExecutionPolicy&& exec,
    RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);
```

1. Effects: Equivalent to: `return is_heap_until(first, last) == last;`

2. Effects: Equivalent to: `return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last) == last;`

3. Effects: Equivalent to: `return is_heap_until(first, last, comp) == last;`

4. Effects: Equivalent to: `return is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last;`
template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
IndirectStrictWeakOrder<projected<I>, Proj>> Comp = ranges::less>
constexpr bool ranges::is_heap(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Proj = identity,
IndirectStrictWeakOrder<projected<I> R>, Proj>> Comp = ranges::less>
constexpr bool ranges::is_heap(I&& r, Comp comp = {}, Proj proj = {});

Effects: Equivalent to: return ranges::is_heap_until(first, last, comp, proj) == last;

template<class RandomAccessIterator>
constexpr RandomAccessIterator
is_heap_until(RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator>
RandomAccessIterator
is_heap_until(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
constexpr RandomAccessIterator
is_heap_until(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
RandomAccessIterator
is_heap_until(ExecutionPolicy&& exec,
RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<RandomAccessIterator I, Sentinel<I> S, class Proj = identity,
IndirectStrictWeakOrder<projected<I>, Proj>> Comp = ranges::less>
constexpr I ranges::is_heap_until(I first, S last, Comp comp = {}, Proj proj = {});
template<RandomAccessRange R, class Proj = identity,
IndirectStrictWeakOrder<projected<I> R>, Proj>> Comp = ranges::less>
safe_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.

25.7.8 Minimum and maximum

template<class T>
constexpr const T& min(const T& a, const T& b);
template<class T, class Compare>
constexpr const T& min(const T& a, const T& b, Compare comp);

template<class T, class Proj = identity,
IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less>
constexpr const T& ranges::min(const T& a, const T& b, Compare comp = {}, Proj proj = {});

Requires: For the first form, type T shall be Cpp17LessThanComparable (Table 24).

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);

§ 25.7.8
template<Copyable T, class Proj = identity,
         IndirectStrictWeakOrder<Projected<arg as const T*, Proj>>> Comp = ranges::less>
constexpr T ranges::min(initializer_list<T> r, Comp comp = {}, Proj proj = {});

template<InputRange R, class Proj = identity,
         IndirectStrictWeakOrder<Projected<iterator_t<R>, Proj>>> Comp = ranges::less>
requires IndirectlyCopyableStorable<iterator_t<R>, Proj>>>
constexpr iter_value_t<iterator_t<R>>>
ranges::min(R&& r, Comp comp = {}, Proj proj = {});

Requires: ranges::distance(r) > 0. For the overloads in namespace std, T shall be Cpp17Copy-
Constructible. For the first form, type T shall be Cpp17LessThanComparable.

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);

requires IndirectlyCopyableStorable<iterator_t<R>, Proj>>>
ranges::max(R&& r, Comp comp = {}, Proj proj = {});

Requires: ranges::distance(r) > 0. For the overloads in namespace std, T shall be Cpp17Copy-
Constructible. For the first form, type T shall be Cpp17LessThanComparable.

Returns: The largest value in the input range.

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, class Proj = identity, 
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less> 
constexpr ranges::minmax_result<const T&> 
ranges::minmax(const T& a, const T& b, Comp comp = {}, Proj proj = {});

17 Requires: For the first form, type T shall be Cpp17LessThanComparable (Table 24).
18 Returns: {b, a} if b is smaller than a, and {a, b} otherwise.
19 Remarks: An invocation may explicitly specify an argument for the template parameter T of the 
overloads in namespace std.
20 Complexity: Exactly one comparison and two applications of the projection, if any.

18 template<class T> 
constexpr pair<T, T> minmax(initializer_list<T> t);
19 template<class T, class Compare> 
constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
20 template<Copyable T, class Proj = identity, 
    IndirectStrictWeakOrder<projected<const T*, Proj>> Comp = ranges::less> 
constexpr ranges::minmax_result<T> 
ranges::minmax(initializer_list<T> r, Comp comp = {}, Proj proj = {});
21 template<InputRange R, class Proj = identity, 
    IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less> 
requires IndirectlyCopyableStorable<iterator_t<R>, iter_value_t<iterator_t<R>>> 
constexpr ranges::minmax_result<iter_value_t<iterator_t<R>>> 
ranges::minmax(R&& r, Comp comp = {}, Proj proj = {});
22 Requires: ranges::distance(r) > 0. For the overloads in namespace std, T shall be Cpp17CopyConstructible. For the first form, type T shall be Cpp17LessThanComparable.
23 Returns: Let X be the return type. Returns Xx, y, where x is a copy of the leftmost element with the 
smallest and y a copy of the rightmost element with the largest value in the input range.
24 Remarks: An invocation may explicitly specify an argument for the template parameter T of the 
overloads in namespace std.
25 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> 
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
26 template<class ExecutionPolicy, class ForwardIterator> 
ForwardIterator min_element(ExecutionPolicy&& exec, 
    ForwardIterator first, ForwardIterator last);
27 template<class ForwardIterator, class Compare> 
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last, 
    Compare comp);
28 template<class ExecutionPolicy, class ForwardIterator, class Compare> 
ForwardIterator min_element(ExecutionPolicy&& exec, 
    ForwardIterator first, ForwardIterator last, 
    Compare comp);
29 template<ForwardIterator I, Sentinel<I> S, class Proj = identity, 
    IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less> 
constexpr I ranges::min_element(I first, S last, Comp comp = {}, Proj proj = {});
30 template<ForwardRange R, class Proj = identity, 
    IndirectStrictWeakOrder<projected<iterator_t<R>, Proj>> Comp = ranges::less> 
constexpr safe_iterator_t<R> 
ranges::min_element(R&& r, Comp comp = {}, Proj proj = {});
31 Let comp be less{} and proj be identity{} for the overloads with no parameters by those names.
32 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.

```cpp
template<class ForwardIterator>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator max_element(ExecutionPolicy&& exec,
                             ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ForwardIterator, class Compare>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
                                      Compare comp);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator max_element(ExecutionPolicy&& exec,
                             ForwardIterator first, ForwardIterator last,
                             Compare comp);
```

```cpp
template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
constexpr I ranges::max_element(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<iterator_t<R>, Proj>>
constexpr safe_iterator_t<R> ranges::max_element(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 first iterator i in the range [first, last) such that for every iterator j in the range [first, last),

bool(invoke(comp, invoke(proj, *i), invoke(proj, *j)))

is false. Returns last if first == last.

**Complexity:** Exactly max(last - first - 1, 0) comparisons and twice as many projections.

```cpp
template<class ForwardIterator>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator>
pair<ForwardIterator, ForwardIterator> minmax_element(ExecutionPolicy&& exec,
                                                      ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ForwardIterator, class Compare>
constexpr pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ExecutionPolicy&& exec,
                                                      ForwardIterator first, ForwardIterator last, Compare comp);
```

```cpp
template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
         IndirectStrictWeakOrder<projected<I, Proj>> Comp = ranges::less>
constexpr ranges::minmax_result<I> ranges::minmax_element(I first, S last, Comp comp = {}, Proj proj = {});
```

```cpp
template<ForwardRange R, class Proj = identity,
         IndirectStrictWeakOrder<iterator_t<R>, Proj>>
constexpr ranges::minmax_result<safe_iterator_t<R>> ranges::minmax_element(R&& r, Comp comp = {}, Proj proj = {});
```

**Returns:** {first, first} if [first, last) is empty, otherwise {m, M}, where m is the first iterator in [first, last) such that no iterator in the range refers to a smaller element, and where M is the last iterator in [first, last) such that no iterator in the range refers to a larger element. This behavior intentionally differs from max_element().

§ 25.7.8
Complexity: Let \( N \) be \( \text{last} - \text{first} \). At most \( \max\left(\left\lfloor \frac{3}{2}(N-1) \right\rfloor, 0\right) \) comparisons and twice as many applications of the projection, if any.

25.7.9 Bounded value

```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);
```

1 Requires: The value of \( \text{lo} \) shall be no greater than \( \text{hi} \). For the first form, type \( T \) shall be \text{Cpp17LessThanComparable} \ (\text{Table 24}).

2 Returns: \( \text{lo} \) if \( v \) is less than \( \text{lo} \), \( \text{hi} \) if \( \text{hi} \) is less than \( v \), otherwise \( v \).

3 [Note: If NaN is avoided, \( T \) can be a floating-point type. — end note]

4 Complexity: At most two comparisons.

25.7.10 Lexicographical comparison

```cpp
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);
```

1 Returns: true if and only if the sequence of elements defined by the range \([\text{first1}, \text{last1})\) is lexicographically less than the sequence of elements defined by the range \([\text{first2}, \text{last2})\).

2 Complexity: At most \( 2 \min(\text{last1} - \text{first1}, \text{last2} - \text{first2}) \) applications of the corresponding comparison and each projection, if any.

3 Remarks: If two sequences have the same number of elements and their corresponding elements (if any) are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a
proper prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

[Example: ranges::lexicographical_compare(I1, S1, I2, S2, Comp, Proj1, Proj2) could be implemented as:

```cpp
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
    if (invoke(comp, invoke(proj1, *first1), invoke(proj2, *first2))) return true;
    if (invoke(comp, invoke(proj2, *first2), invoke(proj1, *first1))) return false;
} return first1 == last1 && first2 != last2;
—end example]

[Note: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. —end note]

25.7.11 Three-way comparison algorithms  
[alg.3way]

```cpp
template<class T, class U> constexpr auto compare_3way(const T& a, const U& b);
```
1 Effects: Compares two values and produces a result of the strongest applicable comparison category type:

(1.1) Returns a <=> b if that expression is well-formed.
(1.2) Otherwise, if the expressions a == b and a < b are each well-formed and convertible to bool, returns strong_ordering::equal when a == b is true, otherwise returns strong_ordering::less when a < b is true, and otherwise returns strong_ordering::greater.
(1.3) Otherwise, if the expression a == b is well-formed and convertible to bool, returns strong_equality::equal when a == b is true, and otherwise returns strong_equality::nonequal.
(1.4) Otherwise, the function is defined as deleted.

```cpp
template<class InputIterator1, class InputIterator2, class Cmp>
constexpr auto lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                                          InputIterator2 b2, InputIterator2 e2,
                                          Cmp comp)
-> common_comparison_category_t<decltype(comp(*b1, *b2)), strong_ordering>;
```
2 Requires: Cmp shall be a function object type whose return type is a comparison category type.
3 Effects: Lexicographically compares two ranges and produces a result of the strongest applicable comparison category type. Equivalent to:

```cpp
for ( ; b1 != e1 && b2 != e2 ; void(++b1), void(++b2) )
    if (auto cmp = comp(*b1,*b2); cmp != 0)
        return cmp;
return b1 != e1 ? strong_ordering::greater :
    b2 != e2 ? strong_ordering::less :
        strong_ordering::equal;
```
4 Equivalence to:

```cpp
return lexicographical_compare_3way(b1, e1, b2, e2,
    [](const auto& t, const auto& u) {
        return compare_3way(t, u);
    });
```

25.7.12 Permutation generators  
[alg.permutation.generators]

```cpp
template<class BidirectionalIterator>
constexpr bool next_permutation(BidirectionalIterator first,
                                BidirectionalIterator last);
```

§ 25.7.12
template<class BidirectionalIterator, class Compare>
constexpr bool next_permutation(BidirectionalIterator first,
    BidirectionalIterator last, Compare comp);

template<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
    class Proj = identity>
requires Sortable<I, Comp, Proj>
constexpr bool
ranges::next_permutation(I first, S last, Comp comp = {}, Proj proj = {});

template<BidirectionalRange R, class Comp = ranges::less,
    class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr bool
ranges::next_permutation(R&& r, Comp comp = {}, Proj proj = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

Requires: For the overloads in namespace std, BidirectionalIterator shall meet the Cpp17Value-
Swappable requirements (16.5.3.2).

Effects: Takes a sequence defined by the range [first, last) and transforms it into the next permut-
ation. 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: true if and only if a next permutation was found.

Complexity: At most (last - 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<BidirectionalIterator I, Sentinel<I> S, class Comp = ranges::less,
    class Proj = identity>
requires Sortable<I, Comp, Proj>
constexpr bool
ranges::prev_permutation(I first, S last, Comp comp = {}, Proj proj = {});

template<BidirectionalRange R, class Comp = ranges::less,
    class Proj = identity>
requires Sortable<iterator_t<R>, Comp, Proj>
constexpr bool
ranges::prev_permutation(R&& r, Comp comp = {}, Proj proj = {});

Let comp be less{} and proj be identity{} for overloads with no parameters by those names.

Requires: For the overloads in namespace std, BidirectionalIterator shall meet the Cpp17Value-
Swappable requirements (16.5.3.2).

Effects: Takes a sequence defined by the range [first, last) and transforms it into the previous permut-
ation. The previous permutation is found by assuming that the set of all permutations is lexicographically
sorted with respect to comp and proj. If no such permutation exists, transforms the sequence into the
last permutation; that is, the descendingly-sorted one.

Returns: true if and only if a previous permutation was found.

Complexity: At most (last - first) / 2 swaps.

25.8 Header <numeric> synopsis

namespace std {
    // 25.9.2, accumulate
    template<class InputIterator, class T>
    T accumulate(InputIterator first, InputIterator last, T init);
template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation binary_op);

// 25.9.3, reduce
template<class InputIterator>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first, InputIterator last);
template<class InputIterator, class T>
T reduce(InputIterator first, InputIterator last, T init);
template<class InputIterator, class T, class BinaryOperation>
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);

// 25.9.4, inner product
template<class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1,
                BinaryOperation2 binary_op2);

// 25.9.5, transform reduce
template<class InputIterator1, class InputIterator2, class T>
T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2, T init);
template<class InputIterator1, class InputIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
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>
T transform_reduce(InputIterator first, InputIterator last,
                   T init,
                   BinaryOperation binary_op, UnaryOperation unary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
T transform_reduce(ExecutionPolicy&& exec, // see 25.3.5
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, T init);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T,
         class BinaryOperation1, class BinaryOperation2>
T transform_reduce(ExecutionPolicy&& exec, // see 25.3.5
                   ForwardIterator1 first1, ForwardIterator1 last1,
                   ForwardIterator2 first2, T init,
                   BinaryOperation1 binary_op1,
template<class ExecutionPolicy,  
    class ForwardIterator, class T,  
    class BinaryOperation, class UnaryOperation>
T transform_reduce(ExecutionPolicy&& exec, // see 25.3.5  
    ForwardIterator first, ForwardIterator last,  
    T init,  
    BinaryOperation binary_op, UnaryOperation unary_op);

// 25.9.6, partial sum
template<class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first,  
    InputIterator last,  
    OutputIterator result);

// 25.9.6, partial sum
template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first,  
    InputIterator last,  
    OutputIterator result,  
    BinaryOperation binary_op);

// 25.9.7, exclusive scan
template<class InputIterator, class OutputIterator, class T>
OutputIterator exclusive_scan(InputIterator first, InputIterator last,  
    OutputIterator result,  
    T init);

// 25.9.7, exclusive scan
template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
OutputIterator exclusive_scan(InputIterator first, InputIterator last,  
    OutputIterator result,  
    T init, BinaryOperation binary_op);

// 25.9.7, exclusive scan
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 25.3.5  
    ForwardIterator1 first, ForwardIterator1 last,  
    ForwardIterator2 result,  
    T init);

// 25.9.7, exclusive scan
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);

// 25.9.8, inclusive scan
template<class InputIterator, class OutputIterator>
OutputIterator inclusive_scan(InputIterator first, InputIterator last,  
    OutputIterator result);

// 25.9.8, inclusive scan
template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator inclusive_scan(InputIterator first, InputIterator last,  
    OutputIterator result,  
    BinaryOperation binary_op);

// 25.9.8, inclusive scan
template<class InputIterator, class OutputIterator, class BinaryOperation, class T>
OutputIterator inclusive_scan(InputIterator first, InputIterator last,  
    OutputIterator result,  
    BinaryOperation binary_op, T init);

// 25.9.8, inclusive scan
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5  
    ForwardIterator1 first, ForwardIterator1 last,  
    ForwardIterator2 result);

// 25.9.8, inclusive scan
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryOperation>
ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5  
    ForwardIterator1 first, ForwardIterator1 last,  
    ForwardIterator2 result,  
    BinaryOperation binary_op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryOperation, class T>
ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
BinaryOperation binary_op, T init);

// 25.9.9, transform exclusive scan
template<class InputIterator, class OutputIterator, class T, class BinaryOperation, class UnaryOperation>
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, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
T init,
BinaryOperation binary_op,
UnaryOperation unary_op);

// 25.9.10, transform inclusive scan
template<class InputIterator, class OutputIterator, class BinaryOperation, class UnaryOperation>
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>
OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
OutputIterator result,
BinaryOperation binary_op,
UnaryOperation unary_op,
T init);

template<class ExecutionPolicy,
class ForwardIterator1, class ForwardIterator2, class BinaryOperation, class UnaryOperation>
ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
BinaryOperation binary_op,
UnaryOperation unary_op);

template<class ExecutionPolicy,
class ForwardIterator1, class ForwardIterator2, class BinaryOperation, class UnaryOperation, class T>
ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, // see 25.3.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
BinaryOperation binary_op,
UnaryOperation unary_op,
T init);

// 25.9.11, adjacent difference
template<class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first,
InputIterator last,
OutputIterator result);
template<class InputIterator, class OutputIterator, class BinaryOperation>
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);

// 25.9.12, iota
template<class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);

// 25.9.13, greatest common divisor
template<class M, class N>
constexpr common_type_t<M, N> gcd(M m, N n);

// 25.9.14, least common multiple
template<class M, class N>
constexpr common_type_t<M, N> lcm(M m, N n);

// 25.9.15, midpoint
template<class T>
constexpr T midpoint(T a, T b) noexcept;

template<class T>
constexpr T* midpoint(T* a, T* b);

25.9 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]

25.9.1 Definitions [numerics.defns]

1 Define GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aN) as follows:

(1.1) — a1 when N is 1, otherwise
(1.2) — op(GENERALIZED_NONCOMMUTATIVE_SUM(op, a1, ..., aK),
GENERALIZED_NONCOMMUTATIVE_SUM(op, aM, ..., aN)) for any K where 1 < K + 1 = M ≤ N.

2 Define GENERALIZED_SUM(op, a1, ..., aN) as GENERALIZED_NONCOMMUTATIVE_SUM(op, b1, ..., bN),
where b1, ..., bN may be any permutation of a1, ..., aN.

25.9.2 Accumulate [accumulate]

template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
BinaryOperation binary_op);

1 Requires: T shall satisfy the Cpp17CopyConstructible (Table 27) and Cpp17CopyAssignable (Table 29)
requirements. In the range [first, last], binary_op shall neither modify elements nor invalidate
iterators or subranges.239

239) The use of fully closed ranges is intentional.
2 Effects: Computes its result by initializing the accumulator acc with the initial value init and then modifies it with
acc = std::move(acc) + *i or acc = binary_op(std::move(acc), *i) for every iterator i in the range [first, last) in order.240

25.9.3 Reduce

```
template<class InputIterator>
typename iterator_traits<InputIterator>::value_type
reduce(InputIterator first, InputIterator last);
```

Effects: Equivalent to:
```
return reduce(first, last,
    typename iterator_traits<InputIterator>::value_type());
```

```
template<class ExecutionPolicy, class ForwardIterator>
typename iterator_traits<ForwardIterator>::value_type
reduce(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last);
```

Effects: Equivalent to:
```
return reduce(std::forward<ExecutionPolicy>(exec), first, last,
    typename iterator_traits<ForwardIterator>::value_type());
```

```
template<class InputIterator, class T>
T reduce(InputIterator first, InputIterator last, T init);
```

Effects: Equivalent to:
```
return reduce(first, last, init, plus<>());
```

```
template<class ExecutionPolicy, class ForwardIterator, class T>
T reduce(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last, T init);
```

Effects: Equivalent to:
```
return reduce(std::forward<ExecutionPolicy>(exec), first, last, init, plus<>());
```

#### Notes

240) accumulate is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.
25.9.4 Inner product

\[
\text{inner_product(class InputIterator1, class InputIterator2, class T)}
\]

T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init);

\[
\text{inner_product(class InputIterator1, class InputIterator2, class T,}
\text{ class BinaryOperation1, class BinaryOperation2)}
\]

T inner_product(InputIterator1 first1, InputIterator1 last1,
                  InputIterator2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);

\[1\]
\text{Requires: T shall satisfy the Cpp17CopyConstructible (Table 27) and Cpp17CopyAssignable (Table 29) requirements. In the ranges [first1, last1] and [first2, first2 + (last1 - first1)] binary_op1 and binary_op2 shall neither modify elements nor invalidate iterators or subranges.}\[241\]

\[2\]
\text{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.}

25.9.5 Transform reduce

\[
\text{transform_reduce(class InputIterator1, class InputIterator2, class T)}
\]

T transform_reduce(InputIterator1 first1, InputIterator1 last1,
                   InputIterator2 first2,
                   T init);

\[1\]
\text{Effects: Equivalent to:}

\[
\text{return transform_reduce(first1, last1, first2, init, plus<>()}, \\text{multiplies<>()};
\]

\[
\text{transform_reduce(ExecutionPolicy&& exec,}
\text{ForwardIterator1 first1, ForwardIterator1 last1,}
\text{ForwardIterator2 first2,}
\text{T init});
\]

\[2\]
\text{Effects: Equivalent to:}

\[
\text{return transform_reduce(std::forward<ExecutionPolicy>(exec),}
\text{first1, last1, first2, init, plus<>()}, \\text{multiplies<>()};
\]

\[
\text{transform_reduce(ExecutionPolicy&& exec,}
\text{ForwardIterator1 first1, ForwardIterator1 last1,}
\text{ForwardIterator2 first2,}
\text{T init,}
\text{BinaryOperation1 binary_op1,}
\text{BinaryOperation2 binary_op2);}\]

\[3\]
\text{Requires:}

\[3.1\]
\text{T shall be Cpp17MoveConstructible (Table 26).}

\[3.2\]
\text{All of}

\[3.2.1\]
\text{— binary_op1(init, init),}

\[241\] The use of fully closed ranges is intentional.
(3.2.2) — binary_op1(init, binary_op2(*first1, *first2)),
(3.2.3) — binary_op1(binary_op2(*first1, *first2), init), and
(3.2.4) — binary_op1(binary_op2(*first1, *first2), binary_op2(*first1, *first2))
shall be convertible to T.
(3.3) — Neither binary_op1 nor binary_op2 shall invalidate subranges, nor modify 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>
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);
```

Requires:
(6.1) — T shall be Cpp17MoveConstructible (Table 26).
(6.2) — All of
(6.2.1) — binary_op(init, init),
(6.2.2) — binary_op(init, unary_op(*first)),
(6.2.3) — binary_op(unary_op(*first), init), and
(6.2.4) — binary_op(unary_op(*first), unary_op(*first))
shall be convertible to T.
(6.3) — Neither unary_op nor binary_op shall invalidate subranges, nor modify elements in the range [first, last].

Returns:
GENERALIZED_SUM(binary_op, init, unary_op(*i), ...)
for every iterator i in [first, last).

Complexity: \( \Theta(last - first) \) applications each of unary_op and binary_op.

[Note: transform_reduce does not apply unary_op to init. — end note]

### 25.9.6 Partial sum

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result);
```

```cpp
template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(
    InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);
```

Requires: InputIterator’s value type shall be constructible from the type of *first. The result of the expression std::move(acc) + *i or binary_op(std::move(acc), *i) shall be implicitly convertible to InputIterator’s value type. acc shall be writable (23.3.1) to the result output iterator. In the
ranges \([\text{first}, \text{last}]\) and \([\text{result}, \text{result} + (\text{last} - \text{first})]\) binary \_op shall neither modify elements nor invalidate iterators or subranges.\(^{242}\)

**Effects:** For a non-empty range, the function creates an accumulator \(\text{acc}\) whose type is \text{InputIterator}'s value type, initializes it with \(*\text{first}\), and assigns the result to \(*\text{result}\). For every iterator \(i\) in \([\text{first} + 1, \text{last})\) in order, \(\text{acc}\) is then modified by \(\text{acc} = \text{std::move(}\text{acc}\text{)} + *i\) or \(\text{acc} = \text{binary\_op(}\text{std::move(}\text{acc}\text{)}, *i)\) and the result is assigned to \(*(\text{result} + (i - \text{first}))\).

**Returns:** \(\text{result} + (\text{last} - \text{first})\).

**Complexity:** Exactly \((\text{last} - \text{first}) - 1\) applications of the binary operation.

**Remarks:** \text{result} may be equal to \text{first}.

25.9.7 Exclusive scan \[\text{exclusive\_scan}\]

\[
\text{template<class InputIterator, class OutputIterator, class T>}\n\text{OutputIterator exclusive\_scan(InputIterator first, InputIterator last,}\n\text{ OutputIterator result, T init);}\]

**Effects:** Equivalent to:

\[
\text{return exclusive\_scan(first, last, result, init, plus<>());}\]

\[
\text{template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>}\n\text{ForwardIterator2 exclusive\_scan(ExecutionPolicy}\&\& \text{ exec,}\n\text{ ForwardIterator1 first, ForwardIterator1 last,}\n\text{ ForwardIterator2 result, T init);}\]

**Effects:** Equivalent to:

\[
\text{return exclusive\_scan(}\text{std::forward<}\text{ExecutionPolicy}\>(\text{exec}),}\n\text{ first, last, result, init, plus<>());}\]

\[
\text{template<class InputIterator, class OutputIterator, class T, class BinaryOperation>}\n\text{OutputIterator exclusive\_scan(InputIterator first, InputIterator last,}\n\text{ OutputIterator result, T init, BinaryOperation binary\_op);}\]

\[
\text{template<class ExecutionPolicy,}\n\text{ class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>}\n\text{ForwardIterator2 exclusive\_scan(ExecutionPolicy}\&\& \text{ exec,}\n\text{ ForwardIterator1 first, ForwardIterator1 last,}\n\text{ ForwardIterator2 result, T init, BinaryOperation binary\_op);}\]

**Requires:**

(3.1) \(- T\) shall be \text{Cpp17MoveConstructible} (Table 26).

(3.2) \(- \text{All of }\text{binary\_op(}\text{init, init}), \text{binary\_op(}\text{init, *first}), \text{and }\text{binary\_op(*first, *first)}\) shall be convertible to \(T\).

(3.3) \(- \text{binary\_op shall neither invalidate iterators or subranges, nor modify elements in the ranges }\\text{[first, last]} \text{or }\text{[result, result + (last - first)]}\).

**Effects:** For each integer \(K\) in \([0, \text{last} - \text{first})\) assigns through \text{result} + \(K\) the value of:

\[
\text{GENERALIZED\_NONCOMMUTATIVE\_SUM}\]

\[
\text{binary\_op, init, *(first + 0), *(first + 1), \ldots, *(first + K - 1)}\]

**Returns:** The end of the resulting range beginning at \text{result}.

**Complexity:** \(\Theta(\text{last} - \text{first})\) applications of \text{binary\_op}.

**Remarks:** \text{result} may be equal to \text{first}.

[\text{Note:} \text{The difference between exclusive\_scan and inclusive\_scan is that exclusive\_scan excludes the }i^{th}\text{ input element from the }i^{th}\text{ sum. If binary\_op is not mathematically associative, the behavior of exclusive\_scan may be nondeterministic.} – end note]
25.9.8 Inclusive scan  [inclusive.scan]

```
template<class InputIterator, class OutputIterator>
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>
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>
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);
```

Requires:

1. If `init` is provided, `T` shall be `Cpp17MoveConstructible` (Table 26); otherwise, `ForwardIterator1`'s value type shall be `Cpp17MoveConstructible`.
2. If `init` is provided, all of `binary_op(init, init), binary_op(init, *first), and binary_op(*first, *first)` shall be convertible to `T`; otherwise, `binary_op(*first, *first)` shall be convertible to `ForwardIterator1`'s value type.
3. `binary_op` shall neither invalidate iterators or subranges, nor modify 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. `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, init, *(first + 0), *(first + 1), ..., *(first + K))`
   if `init` is provided, or
2. `GENERALIZED_NONCOMMUTATIVE_SUM(binary_op, *(first + 0), *(first + 1), ..., *(first + K))`
   otherwise.

Returns: The end of the resulting range beginning at `result`.

Complexity: \(O(last - first)\) 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]

25.9.9 Transform exclusive scan  [transform.exclusive.scan]

```
template<class InputIterator, class OutputIterator, class T, class BinaryOperation, class UnaryOperation>
```
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);

1 Requires:
(1.1) — T shall be Cpp17MoveConstructible (Table 26).
(1.2) — All of
(1.2.1) — binary_op(init, init),
(1.2.2) — binary_op(init, unary_op(*first)), and
(1.2.3) — binary_op(unary_op(*first), unary_op(*first))
shall be convertible to T.
(1.3) — Neither unary_op nor binary_op shall invalidate iterators or subranges, nor modify elements in the ranges [first, last] or [result, result + (last - first)].

2 Effects: For each integer K in [0, last - first) assigns through result + K the value of:
   GENERALIZED_NONCOMMUTATIVE_SUM
      binary_op, init, 
      unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K - 1))

3 Returns: The end of the resulting range beginning at result.

4 Complexity: \(O(last - first)\) applications each of unary_op and binary_op.

5 Remarks: result may be equal to first.

6 [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]

25.9.10 Transform inclusive scan [transform.inclusive.scan]

template<class InputIterator, class OutputIterator, 
    class BinaryOperation, class UnaryOperation>
OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last, 
    OutputIterator result, 
    BinaryOperation binary_op, UnaryOperation unary_op);

template<class ExecutionPolicy, 
    class ForwardIterator1, class ForwardIterator2, class T, 
    class BinaryOperation, class UnaryOperation>
ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec, 
    ForwardIterator1 first, ForwardIterator1 last, 
    ForwardIterator2 result, T init, 
    BinaryOperation binary_op, UnaryOperation unary_op);

template<class InputIterator, class OutputIterator, 
    class BinaryOperation, class UnaryOperation, class T>
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 T, 
    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);
BinaryOperation binary_op, UnaryOperation unary_op, T init);

1

Requires:

(1.1) If init is provided, T shall be Cpp17MoveConstructible (Table 26); otherwise, ForwardIterator1’s value type shall be Cpp17MoveConstructible.

(1.2) If init is provided, all of

(1.2.1) binary_op(init, init),

(1.2.2) binary_op(init, unary_op(*first)), and

(1.2.3) binary_op(unary_op(*first), unary_op(*first))

shall be convertible to T; otherwise, binary_op(unary_op(*first), unary_op(*first)) shall be convertible to ForwardIterator1’s value type.

(1.3) Neither unary_op nor binary_op shall invalidate iterators or subranges, nor modify 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

(2.1) GENERALIZED_NONCOMMUTATIVE_SUM(
        binary_op, init,
        unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K)))

if init is provided, or

(2.2) GENERALIZED_NONCOMMUTATIVE_SUM(
        binary_op,
        unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K)))

otherwise.

Returns: The end of the resulting range beginning at result.

Complexity: $O((last - first))$ 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_inclusive_scan includes the $i$th input element in the $i$th sum. If binary_op is not mathematically associative, the behavior of transform_inclusive_scan may be nondeterministic. transform_inclusive_scan does not apply unary_op to init. — end note]

25.9.11 Adjacent difference [adjacent.difference]

template<class InputIterator, class OutputIterator>
OutputIterator
adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
adjacent_difference(ExecutionPolicy&& exec,
                   ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator
adjacent_difference(InputIterator first, InputIterator last,
                    OutputIterator result, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryOperation>
ForwardIterator2
adjacent_difference(ExecutionPolicy&& exec,
                    ForwardIterator1 first, ForwardIterator1 last,
                    ForwardIterator2 result, BinaryOperation binary_op);

1

Let T be the value type of decltype(first). For the overloads that do not take an argument binary_op, let binary_op be an lvalue that denotes an object of type minus<>.

2

Requires:
— For the overloads with no ExecutionPolicy, $T$ shall be $Cpp17MoveAssignable$ (Table 28) and shall be constructible from the type of $*\text{first}$. $\text{acc}$ (defined below) shall be writable (23.3.1) to the $\text{result}$ output iterator. The result of the expression $\text{binary\_op}(\text{val}, \text{std}::\text{move}(\text{acc}))$ shall be writable to the $\text{result}$ output iterator.

— For the overloads with an ExecutionPolicy, the result of the expressions $\text{binary\_op}(\text{first}, \text{first})$ and $\text{first}$ shall be writable to $\text{result}$.

— For all overloads, in the ranges $[\text{first}, \text{last}]$ and $[\text{result}, \text{result} + (\text{last} - \text{first})]$, $\text{binary\_op}$ shall neither modify elements nor invalidate iterators or subranges.

**Effects:** For the overloads with no ExecutionPolicy and a non-empty range, the function creates an accumulator $\text{acc}$ of type $T$, initializes it with $*\text{first}$, and assigns the result to $*\text{result}$. For every iterator $i$ in $[\text{first} + 1, \text{last})$ in order, creates an object $\text{val}$ whose type is $T$, initializes it with $*i$, computes $\text{binary\_op}(\text{val}, \text{std}\cdot\text{move}(\text{acc}))$, assigns the result to $*(\text{result} + (i - \text{first}))$, and move assigns from $\text{val}$ to $\text{acc}$.

For the overloads with an ExecutionPolicy and a non-empty range, performs $*\text{result} = *\text{first}$. Then, for every $d$ in $[1, \text{last} - \text{first} - 1]$, performs $*(\text{result} + d) = \text{binary\_op}(*(\text{first} + d), *(\text{first} + (d - 1)))$.

**Returns:** $\text{result} + (\text{last} - \text{first})$.

**Complexity:** Exactly $(\text{last} - \text{first}) - 1$ applications of the binary operation.

**Remarks:** For the overloads with no ExecutionPolicy, $\text{result}$ may be equal to $\text{first}$. For the overloads with an ExecutionPolicy, the ranges $[\text{first}, \text{last})$ and $[\text{result}, \text{result} + (\text{last} - \text{first})]$ shall not overlap.

### 25.9.12 Iota [numeric.iota]

```cpp
template<class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

1 **Requires:** $T$ shall be convertible to $\text{ForwardIterator}$’s value type. The expression $++\text{val}$, where $\text{val}$ has type $T$, shall be well-formed.

2 **Effects:** For each element referred to by the iterator $i$ in the range $[\text{first}, \text{last})$, assigns $*i = \text{value}$ and increments $\text{value}$ as if by $++\text{value}$.

3 **Complexity:** Exactly $\text{last} - \text{first}$ increments and assignments.

### 25.9.13 Greatest common divisor [numeric.ops.gcd]

```cpp
template<class M, class N>
constexpr common_type_t<M, N> gcd(M m, N n);
```

1 **Requires:** $|m|$ and $|n|$ shall be representable as a value of $\text{common\_type\_t}_t<M, N>$. [Note: These requirements ensure, for example, that $\gcd(m, m) = |m|$ is representable as a value of type $M$. — end note]

2 **Remarks:** If either $M$ or $N$ is not an integer type, or if either is $\text{cv bool}$, the program is ill-formed.

3 **Returns:** Zero when $m$ and $n$ are both zero. Otherwise, returns the greatest common divisor of $|m|$ and $|n|$.

4 **Throws:** Nothing.

### 25.9.14 Least common multiple [numeric.ops.lcm]

```cpp
template<class M, class N>
constexpr common_type_t_t<M, N> lcm(M m, N n);
```

1 **Requires:** $|m|$ and $|n|$ shall be representable as a value of $\text{common\_type\_t}_t<M, N>$. The least common multiple of $|m|$ and $|n|$ shall be representable as a value of type $\text{common\_type\_t}_t<M, N>$.

2 **Remarks:** If either $M$ or $N$ is not an integer type, or if either is $\text{cv bool}$ the program is ill-formed.

3 **Returns:** Zero when either $m$ or $n$ is zero. Otherwise, returns the least common multiple of $|m|$ and $|n|$.

4 **Throws:** Nothing.

243 The use of fully closed ranges is intentional.
25.9.15 Midpoint

```cpp
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.

```cpp
template<class T>
constexpr T* midpoint(T* a, T* b);
```

4. **Constraints:** T is a complete object type.
5. **Expects:** a and b point to, respectively, elements x[i] and x[j] of the same array object x. [Note: An object that is not an array element is considered to belong to a single-element array for this purpose; see 7.6.2.1. A pointer past the last element of an array x of n elements is considered to be equivalent to a pointer to a hypothetical element x[n] for this purpose; see 6.7.2. —end note]
6. **Returns:** A pointer to x[i + \frac{j-i}{2}], where the result of the division is truncated towards zero.

25.10 C library algorithms

```cpp
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);
```

1. **Effects:** These functions have the semantics specified in the C standard library.
2. **Remarks:** The behavior is undefined unless the objects in the array pointed to by base are of trivial type.
3. **Throws:** Any exception thrown by compar() (16.5.5.12).

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 functions for floating-point types, as summarized in Table 81.

Table 81 — Numerics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.2</td>
<td>Requirements</td>
</tr>
</tbody>
</table>
| 26.3      | 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 |

26.2 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 satisfies 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` satisfies 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

```c
#define FE_ALL_EXCEPT see below  // optional
#define FE_DVBYZERO 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
```

244) In other words, value types. These include arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.
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);
}

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 (15.8) 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]

The floating-point environment has thread storage duration (6.6.5.2). 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 (32.3.2) 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]

A separate floating-point environment shall be 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]

The header `<complex>` defines a class template, and numerous functions for representing and manipulating complex numbers.

The effect of instantiating the template `complex` for any type other than `float`, `double`, or `long double` is unspecified. The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types (6.7).

If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

If `z` is an lvalue of type `cv complex<T>` then:

(4.1) the expression `reinterpret_cast<cv T&>[2]>(z)` shall be well-formed,

(4.2) `reinterpret_cast<cv T&>[2]>(z)[0]` shall designate the real part of `z`, and

(4.3) `reinterpret_cast<cv T&>[2]>(z)[1]` shall designate 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]` shall designate the real part of `a[i]`, and

(4.5) `reinterpret_cast<cv T*>(a)[2*i + 1]` shall designate the imaginary part of `a[i]`.

26.4.1 Header `<complex>` synopsis [complex.syn]

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>&);

// 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>&);

// 26.4.8, transcendentals
template<class T> complex<T> acos(const complex<T>&);
template<class T> complex<T> asin(const complex<T>&);

// 26.4.1096
template<class T> complex<T> log10(const complex<T>&);

template<class T> complex<T> pow (const complex<T>&, const 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>&);
    }
}

1 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;
            § 26.4.3 1097
constexpr complex(float re = 0.0f, float im = 0.0f);
constexpr complex(const complex<float>&) = default;
constexpr explicit complex(const complex<double>&);
constexpr explicit complex(const complex<long double>&);

constexpr float real() const;
constexpr void real(float);
constexpr float imag() const;
constexpr void imag(float);

constexpr complex& operator=(float);
constexpr complex& operator+=(float);
constexpr complex& operator-=(float);
constexpr complex& operator*=(float);
constexpr complex& operator/=(float);

constexpr complex& operator=(const complex&);
template<class X> constexpr complex& operator=(const complex<X>&);

template<> class complex<double> {
public:
    using value_type = double;

    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&) = default;
    constexpr explicit complex(const complex<long double>&);

    constexpr double real() const;
    constexpr void real(double);
    constexpr double imag() const;
    constexpr void imag(double);

    constexpr complex& operator=(double);
    constexpr complex& operator+=(double);
    constexpr complex& operator-=(double);
    constexpr complex& operator*=(double);
    constexpr complex& operator/=(double);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator=(const complex<X>&);
};

template<> class complex<long double> {
public:
    using value_type = long double;

    constexpr complex(long double re = 0.0L, long double im = 0.0L);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&);
    constexpr complex(const complex<long double>&) = default;

    constexpr long double real() const;
    constexpr void real(long double);
    constexpr long double imag() const;
    constexpr void imag(long double);

};
constexpr void imag(long double);
constexpr complex& operator=(long double);
constexpr complex& operator+=(long double);
constexpr complex& operator-=(long double);
constexpr complex& operator*=(long double);
constexpr complex& operator/=(long double);
constexpr complex& operator=(const complex&);
template<class X> constexpr complex& operator=(const complex<X>&);
template<class X> constexpr complex& operator+=(const complex<X>&);
template<class X> constexpr complex& operator-=(const complex<X>&);
template<class X> constexpr complex& operator*=(const complex<X>&);
template<class X> constexpr complex& operator/=(const complex<X>&);
}

26.4.4 Member functions

template<class T> constexpr complex(const T& re = T(), const T& im = T());

Effects: Constructs an object of class complex.
Ensures: real() == re && imag() == im.

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

customexpr 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.
customexpr 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.
customexpr complex& operator*=(const T& rhs);
Effects: Multiplies the scalar value rhs by the complex value *this and stores the result in *this.
Returns: *this.
customexpr 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> customexpr 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 [complex.ops]

template<class T> constexpr complex<T> operator+(const complex<T>& lhs);
  
  Returns: complex<T>(lhs).
  
  Remarks: unary operator.

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);
  
  Returns: complex<T>(-lhs.real(),-lhs.imag()).
  
  Remarks: unary operator.

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);

template<class T> constexpr bool operator==(const T& lhs, const complex<T>& rhs);
  
  Returns: lhs.real() == rhs.real() && lhs.imag() == rhs.imag().

template<class T, class charT, class traits>
  
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& is, complex<T>& x);
  
  Requires: The input values shall be convertible to T.
Effects: Extracts a complex number \( x \) of the form: \( u, (u), \) 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::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:

```
basic_ostream<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());
```

Requires: \( \rho \) shall be non-negative and non-NaN. \( \theta \) shall be finite.

Returns: The complex value corresponding to a complex number whose magnitude is \( \rho \) and whose phase angle is \( \theta \).

### 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 \texttt{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 \texttt{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 \( x \), imag(log(x)) lies in the interval \([-\pi, \pi]\). [Note: The semantics of this function are intended to be the same in C++ as they are for 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 \( \exp(y \cdot \log(x)) \). The value returned for \texttt{pow}(0, 0) is implementation-defined.

Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> sin(const complex<T>& x);

Returns: The complex sine of \( x \).

template<class T> complex<T> sinh(const complex<T>& x);

Returns: The complex hyperbolic sine of \( x \).

template<class T> complex<T> sqrt(const complex<T>& x);

Returns: The complex square root of \( x \), in the range of the right half-plane. [Note: The semantics of this function are intended to be the same in C++ as they are for \texttt{csqrt} in C. — end note]
25
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:

\begin{itemize}
  \item \texttt{arg}\n  \item \texttt{conj}\n  \item \texttt{imag}\n  \item \texttt{real}
\end{itemize}

where \texttt{norm}, \texttt{conj}, \texttt{imag}, and \texttt{real} are \texttt{constexpr} overloads.

2 The additional overloads shall be sufficient to ensure:

\begin{enumerate}
  \item— If the argument has type \texttt{long double}, then it is effectively cast to \texttt{complex<long double>}.
  \item— Otherwise, if the argument has type \texttt{double} or an integer type, then it is effectively cast to \texttt{complex<double>}.
  \item— Otherwise, if the argument has type \texttt{float}, then it is effectively cast to \texttt{complex<float>}.
\end{enumerate}

3 Function template \texttt{pow} shall have additional overloads sufficient to ensure, for a call with at least one argument of type \texttt{complex<T>}:

\begin{enumerate}
  \item— If either argument has type \texttt{complex<long double>} or \texttt{long double}, then both arguments are effectively cast to \texttt{complex<long double>}.
  \item— Otherwise, if either argument has type \texttt{complex<double>}, \texttt{double}, or an integer type, then both arguments are effectively cast to \texttt{complex<double>}.
  \item— Otherwise, if either argument has type \texttt{complex<float>} or \texttt{float}, then both arguments are effectively cast to \texttt{complex<float>}.
\end{enumerate}

26.4.10 Suffixes for complex number literals

1 This subclause describes literal suffixes for constructing complex number literals. The suffixes \texttt{i}, \texttt{il}, and \texttt{if} create complex numbers of the types \texttt{complex<double>}, \texttt{complex<long double>}, and \texttt{complex<float>} respectively, with their imaginary part denoted by the given literal number and the real part being zero.

\begin{itemize}
  \item \texttt{constexpr complex<long double> operator""il(long double d);}
  \item \texttt{constexpr complex<long double> operator""il(unsigned long long d);}
\end{itemize}

2 Returns: \texttt{complex<long double>{0.0L, static_cast<long double>(d)}}.

\begin{itemize}
  \item \texttt{constexpr complex<double> operator""i(long double d);}
  \item \texttt{constexpr complex<double> operator""i(unsigned long long d);}
\end{itemize}

3 Returns: \texttt{complex<double>{0.0, static_cast<double>(d)}}.

\begin{itemize}
  \item \texttt{constexpr complex<float> operator""if(long double d);}
  \item \texttt{constexpr complex<float> operator""if(unsigned long long d);}
\end{itemize}

4 Returns: \texttt{complex<float>{0.0f, static_cast<float>(d)}}.

26.5 Bit manipulation

26.5.1 General

1 The header \texttt{<bit>} provides components to access, manipulate and process both individual bits and bit sequences.

26.5.2 Header \texttt{<bit>} synopsis

\begin{verbatim}
namespace std {
  // 26.5.3, bit_cast
  template<
type_name To, type_name From>
  constexpr To bit_cast(const From& from) noexcept;
\end{verbatim}
26.5.4 Integral powers of 2

```
// 26.5.4, integral powers of 2
template<class T>
  constexpr bool ispow2(T x) noexcept;
```

```
template<class T>
  constexpr T ceil2(T x) noexcept;
```

```
template<class T>
  constexpr T floor2(T x) noexcept;
```

```
template<class T>
  constexpr T log2p1(T x) noexcept;
```

26.5.3 Function template bit_cast

```
template<typename To, typename From>
  constexpr To bit_cast(const From& from) noexcept;
```

1 Returns: An object of type To. 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 To object are unspecified. If there is no value of type To corresponding to the value representation produced, the behavior is undefined. If there are multiple such values, which value is produced is unspecified.

2 Remarks: This function shall not participate in overload resolution unless:

1. sizeof(To) == sizeof(From) is true;
2. is_trivially_copyable_v<To> is true; and
3. is_trivially_copyable_v<From> is true.

This function shall be constexpr if and only if To, From, and the types of all subobjects of To and From are types T such that:

4. is_union_v<T> is false;
5. is_pointer_v<T> is false;
6. is_member_pointer_v<T> is false;
7. is_volatile_v<T> is false; and
8. T has no non-static data members of reference type.

26.5.4 Integral powers of 2

```
template<class T>
  constexpr bool ispow2(T x) noexcept;
```

1 Returns: true if x is an integral power of two; false otherwise.

2 Remarks: This function shall not participate in overload resolution unless T is an unsigned integer type (6.7.1).

```
template<class T>
  constexpr T ceil2(T x) noexcept;
```

3 Returns: The minimal value y such that ispow2(y) is true and y >= x; if y is not representable as a value of type T, the result is an unspecified value.

4 Remarks: This function shall not participate in overload resolution unless T is an unsigned integer type (6.7.1).

```
template<class T>
  constexpr T floor2(T x) noexcept;
```

5 Returns: If x == 0, 0; otherwise the maximal value y such that ispow2(y) is true and y <= x.

6 Remarks: This function shall not participate in overload resolution unless T is an unsigned integer type (6.7.1).

```
template<class T>
  constexpr T log2p1(T x) noexcept;
```

7 Returns: If x == 0, 0; otherwise one plus the base-2 logarithm of x, with any fractional part discarded.
Remarks: This function shall not participate in overload resolution unless \( T \) is an unsigned integer type (6.7.1).

### 26.6 Random number generation

This subclause defines a facility for generating (pseudo-)random numbers.

In addition to a few utilities, four categories of entities are described: uniform random bit generators, random number engines, random number engine adaptors, and random number distributions. These categorizations are applicable to types that satisfy 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 \( \text{bind}(d,e) \). — end note]

Each of the entities specified via this subclause has an associated arithmetic type (6.7.1) identified as `result_type`. With \( T \) as the `result_type` thus associated with such an entity, that entity is characterized:

1. as boolean or equivalently as boolean-valued, if \( T \) is \( \text{bool} \);
2. otherwise as integral or equivalently as integer-valued, if \( \text{numeric_limits}<T>::\text{is_integer} \) is true;
3. otherwise as floating or equivalently as real-valued.

If integer-valued, an entity may optionally be further characterized as signed or unsigned, according to \( \text{numeric_limits}<T>::\text{is_signed} \).

Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

Throughout this subclause, the operators `bitand`, `bitor`, and `xor` denote the respective conventional bitwise operations. Further:

- the operator `rshift` denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and
- the operator `lshift` 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

```cpp
#include <initializer_list>
namespace std {
    // 26.6.2.3, uniform random bit generator requirements
    template<class G>  
        concept UniformRandomBitGenerator = see below;

    // 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 s, 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;
26.6.2 Requirements

26.6.2.1 General requirements

Throughout this subclause 26.6, the effect of instantiating a template:

(1.1) — that has a template type parameter named Sseq is undefined unless the corresponding template argument is cv-unqualified and satisfies 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 satisfies 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 satisfies 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.
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 satisfying 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 satisfies 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 $S$ satisfies the requirements of a seed sequence if the expressions shown in Table 82 are valid and have the indicated semantics, and if $S$ also satisfies all other requirements of this subclause 26.6.2.2. In that Table and throughout this subclause:

1. $T$ is the type named by $S$’s associated result_type;
2. $q$ is a value of $S$ and $r$ is a possibly const value of $S$;
3. $ib$ and $ie$ are input iterators with an unsigned integer value_type of at least 32 bits;
4. $rb$ and $re$ are mutable random access iterators with an unsigned integer value_type of at least 32 bits;
5. $ob$ is an output iterator; and
6. $il$ is a value of initializer_list$<$T$>$.

#### Table 82 — Seed sequence requirements

<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 (6.7.1) 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 $S(il.begin(), il.end())$.</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>
</tbody>
</table>
Table 82 — Seed sequence requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>r.param(ob)</td>
<td>void</td>
<td>Copies to the given destination</td>
<td>$O(r.size())$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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></td>
</tr>
</tbody>
</table>

26.6.2.3 Uniform random bit generator requirements

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]

```
template<class G>
concept UniformRandomBitGenerator =
    Invocable<G&> && UnsignedIntegral<invoke_result_t<G&>> &&
    requires {
        { G::min() } -> Same<invoke_result_t<G&>>;
        { G::max() } -> Same<invoke_result_t<G&>>;
    };
```

2 Let $g$ be an object of type $G$. $G$ models **UniformRandomBitGenerator** only if

(2.1) — both $G::$_min() and $G::$_max() are constant expressions (7.7),
(2.2) — $G::$_min() < G::$_max()$,
(2.3) — $G::$_min() <= g()$,
(2.4) — $g() <= G::$_max()$, and
(2.5) — $g()$ has amortized constant complexity.

3 A class $G$ meets the **uniform random bit generator** requirements if $G$ models **UniformRandomBitGenerator**, $invoke_result_t<G&>$ is an unsigned integer type (6.7.1), 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

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.

2 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>>.

3 $E$’s specification shall define:

(3.1) — the size of $E$’s state in multiples of the size of **result_type**, given as an integral constant expression;
(3.2) — the **transition algorithm** $TA$ by which $e$’s state $e_i$ is advanced to its successor state $e_{i+1}$; and
(3.3) — the **generation algorithm** $GA$ by which an engine’s state is mapped to a value of type **result_type**.

4 A class $E$ that satisfies the requirements of a uniform random bit generator (26.6.2.3) also satisfies the requirements of a random number engine if the expressions shown in Table 83 are valid and have the indicated semantics, and if $E$ also satisfies all other requirements of this subclause 26.6.2.4. In that Table and throughout this subclause:

(4.1) — $T$ is the type named by $E$’s associated **result_type**;
(4.2) — $e$ is a value of $E$, $v$ is an lvalue of $E$, $x$ and $y$ are (possibly const) values of $E$;
(4.3) — $z$ is a value of $T$;
(4.4) — $q$ is an lvalue satisfying the requirements of a seed sequence (26.6.2.2);
(4.5) — $z$ is a value of type unsigned long long.
(4.6) — `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>;` and
(4.7) — `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 83 — Random number engine requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E()</code></td>
<td></td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type <code>E</code>.</td>
<td><code>O(size of state)</code></td>
</tr>
<tr>
<td><code>E(x)</code></td>
<td></td>
<td>Creates an engine that compares equal to <code>x</code>.</td>
<td><code>O(size of state)</code></td>
</tr>
<tr>
<td><code>E(s)</code></td>
<td></td>
<td>Creates an engine with initial state determined by <code>s</code>.</td>
<td><code>O(size of state)</code></td>
</tr>
<tr>
<td><code>E(q)</code></td>
<td></td>
<td>Creates an engine with an initial state that depends on a sequence produced by one call to <code>q.generate</code>.</td>
<td>same as complexity of <code>q.generate</code> called on a sequence whose length is size of state</td>
</tr>
<tr>
<td><code>e.seed()</code></td>
<td><code>void</code></td>
<td>Ensures: <code>e == E()</code>.</td>
<td>same as <code>E()</code></td>
</tr>
<tr>
<td><code>e.seed(s)</code></td>
<td><code>void</code></td>
<td>Ensures: <code>e == E(s)</code>.</td>
<td>same as <code>E(s)</code></td>
</tr>
<tr>
<td><code>e.seed(q)</code></td>
<td><code>void</code></td>
<td>Ensures: <code>e == E(q)</code>.</td>
<td>same as <code>E(q)</code></td>
</tr>
<tr>
<td><code>e()</code></td>
<td><code>T</code></td>
<td>Advances <code>e</code>’s state to <code>e_{i+1} = TA(e_i)</code> and returns <code>GA(e_i)</code>.</td>
<td>per 26.6.2.3</td>
</tr>
<tr>
<td><code>e.discard(z)</code></td>
<td><code>void</code></td>
<td>Advances <code>e</code>’s state to <code>e_{i+1}</code> by any means equivalent to <code>z</code> consecutive calls <code>e()</code>.</td>
<td>no worse than the complexity of <code>z</code> consecutive calls <code>e()</code></td>
</tr>
<tr>
<td><code>x == y</code></td>
<td><code>bool</code></td>
<td>This operator is an equivalence relation. With <code>S_x</code> and <code>S_y</code> 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 <code>S_x = S_y</code>; else returns <code>false</code>.</td>
<td><code>O(size of state)</code></td>
</tr>
<tr>
<td><code>x != y</code></td>
<td><code>bool</code></td>
<td><code>!(x == y)</code>.</td>
<td><code>O(size of state)</code></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 <code>ios_base::dec</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. Ensures: The <code>os.fmtflags</code> and fill character are unchanged.</td>
<td><code>O(size of state)</code></td>
</tr>
</tbody>
</table>

245) This constructor (as well as the subsequent corresponding `seed()` function) may be particularly useful to applications requiring a large number of independent random sequences.
246) 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
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<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::failbit)</code> (which may throw <code>ios::failure</code> (29.5.5.4)). 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>Requires:</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>Ensures:</strong> The <code>is.fmtflags</code> are unchanged.</td>
<td>(O(\text{size of state}))</td>
</tr>
</tbody>
</table>

5 E shall satisfy the `Cpp17CopyConstructible` (Table 27) and `Cpp17CopyAssignable` (Table 29) requirements. These operations shall each be of complexity no worse than \(O(\text{size of state})\).

26.6.2.5 Random number engine adaptor requirements

1 A random number engine adaptor (commonly shortened to adaptor) `a` of type `A` is a random number engine that takes values produced by some other random number engine, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. An engine `b` of type `B` adapted in this way is termed a base engine in this context. The expression `a.base()` shall be valid and shall return a const reference to `a`’s base engine.

2 The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

\[\text{A::A();}\]

3 **Effects:** The base engine is initialized as if by its default constructor.

\[\text{bool operator==(const A& a1, const A& a2);}\]

4 **Returns:** `true` if `a1`’s base engine is equal to `a2`’s base engine. Otherwise returns `false`.

\[\text{A::A(result_type s);}\]

5 **Effects:** The base engine is initialized with `s`.

\[\text{template<class Sseq> A::A(Sseq& q);}\]

6 **Effects:** The base engine is initialized with `q`.

\[\text{void seed();}\]

7 **Effects:** With `b` as the base engine, invokes `b.seed()`.

§ 26.6.2.5
### §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 \( D \) satisfies the requirements of a random number distribution if the expressions shown in Table 84 are valid and have the indicated semantics, and if \( D \) and its associated types also satisfy all other requirements of this subclause 26.6.2.6. In that Table and throughout this subclause,

- \( T \) is the type named by \( D \)'s associated result_type;
- \( P \) is the type named by \( D \)'s associated param_type;
- \( d \) is a value of \( D \), and \( x \) and \( y \) are (possibly const) values of \( D \);
- \( \text{glb} \) 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;
- \( p \) is a (possibly const) value of \( P \);
- \( g, g1, \text{ and } g2 \) are lvalues of a type satisfying the requirements of a uniform random bit generator (26.6.2.3);
- \( os \) 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 84 — 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>( D::\text{result} _\text{type} )</td>
<td>( T )</td>
<td>( T ) is an arithmetic type (6.7.1). compile-time</td>
<td></td>
</tr>
<tr>
<td>( D::\text{param} _\text{type} )</td>
<td>( P )</td>
<td>Create a distribution whose behavior is indistinguishable from that of any other newly default-constructed distribution of type ( D ).</td>
<td>constant</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Pre/post-condition</td>
<td>Complexity</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>D(p)</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 p.</td>
<td>same as p’s construction</td>
</tr>
<tr>
<td>d.reset()</td>
<td>void</td>
<td>Subsequent uses of d do not depend on values produced by any engine prior to invoking reset.</td>
<td>constant</td>
</tr>
<tr>
<td>x.param()</td>
<td>P</td>
<td>Returns a value p such that D(p).param() == p.</td>
<td>no worse than the complexity of D(p)</td>
</tr>
<tr>
<td>d.param(p)</td>
<td>void</td>
<td>Ensures: d.param() == p.</td>
<td>no worse than the complexity of D(p)</td>
</tr>
<tr>
<td>d(g)</td>
<td>T</td>
<td>With p = d.param(), the sequence of numbers returned by successive invocations with the same object g is randomly distributed according to the associated p(z</td>
<td>{p}) or P(z_i</td>
</tr>
<tr>
<td>d(g,p)</td>
<td>T</td>
<td>The sequence of numbers returned by successive invocations with the same objects g and p is randomly distributed according to the associated p(z</td>
<td>{p}) or P(z_i</td>
</tr>
<tr>
<td>x.min()</td>
<td>T</td>
<td>Returns g1b.</td>
<td>constant</td>
</tr>
<tr>
<td>x.max()</td>
<td>T</td>
<td>Returns lub.</td>
<td>constant</td>
</tr>
<tr>
<td>x == y</td>
<td>bool</td>
<td>This operator is an equivalence relation. Returns true if x.param() == y.param() and S_1 = S_2, where S_1 and S_2 are the infinite sequences of values that would be generated, respectively, by repeated future calls to x(g1) and y(g2) whenever g1 == g2. Otherwise returns false.</td>
<td>constant</td>
</tr>
<tr>
<td>x != y</td>
<td>bool</td>
<td>!(x == y).</td>
<td>same as x == y.</td>
</tr>
<tr>
<td>os &lt;&lt; x</td>
<td></td>
<td>reference to the type of os</td>
<td>Writes to os a textual representation for the parameters and the additional internal data of x. Ensures: The os.fmtflags and fill character are unchanged.</td>
</tr>
</tbody>
</table>
## Expression | Return type | Pre/post-condition | Complexity
--- | --- | --- | ---
`is >> d` | reference to the type of `is` | Restores from `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 `is.setstate(ios::failbit)` (which may throw `ios::failure` (29.5.5.4)). | Requires: `is` provides a textual representation that was previously written using an `os` whose imbued locale and whose type’s template specialization arguments `charT` and `traits` were the same as those of `is`. Ensures: The `is.fmtflags` are unchanged.

4. `D` shall satisfy the `Cpp17CopyConstructible` (Table 27) and `Cpp17CopyAssignable` (Table 29) requirements.

5. The sequence of numbers produced by repeated invocations of `d(g)` shall be independent of any invocation of `os << d` or of any `const` member function of `D` between any of the invocations `d(g)`.

6. If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(g)` shall produce the same sequence of numbers as would repeated invocations of `x(g)`.

7. It is unspecified whether `D::param_type` is declared as a (nested) `class` or via a `typedef`. In this subclause 26.6, declarations of `D::param_type` are in the form of `typedefs` for convenience of exposition only.

8. `P` shall satisfy the `Cpp17CopyConstructible` (Table 27), `Cpp17CopyAssignable` (Table 29), and `Cpp17Equality-Comparable` (Table 23) requirements.

9. For each of the constructors of `D` taking arguments corresponding to parameters of the distribution, `P` shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `D` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

10. `P` shall have a declaration of the form
    ```
    using distribution_type = D;
    ```

### 26.6.3 Random number engine class templates

Each type instantiated from a class template specified in this subclause 26.6.3 satisfies the requirements of a random number engine (26.6.2.4) type.

Except where specified otherwise, the complexity of each function specified in this subclause 26.6.3 is constant.

Except where specified otherwise, no function described in this subclause 26.6.3 throws an exception.

Every function described in this subclause 26.6.3 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 26.6.3 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.

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.
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 \(GA(x_i) = x_i + 1\).

```cpp
template<class UIntType, UIntType a, UIntType c, UIntType m>
class linear_congruential_engine {
public:
    // types
    using result_type = UIntType;

    // engine characteristics
    static constexpr result_type multiplier = a;
    static constexpr result_type increment = c;
    static constexpr result_type modulus = m;
    static constexpr result_type min() { return c == 0u ? 1u: 0u; }
    static constexpr result_type max() { return m - 1u; }
    static constexpr result_type default_seed = 1u;

    // constructors and seeding functions
    explicit 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);
};
```

1 If the template parameter \(m\) is 0, the modulus \(m\) used throughout this subclause 26.6.3.1 is `numeric_limits<result_type>::max()` plus 1. [Note: \(m\) need not be representable as a value of type `result_type`. —end note]

2 If the template parameter \(m\) is not 0, the following relations shall hold: \(a < m\) and \(c < m\).

3 The textual representation consists of the value of \(x_i\).

4 The engine’s state to 1, otherwise sets the engine’s state to \(s \mod m\).

Effect: Constructs a `linear_congruential_engine` object. 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 \(s \mod m\).
template<class Sseq> explicit linear_congruential_engine(Sseq& q);

Effects: Constructs a linear_congruential_engine object. With \( k = \lceil \log_2 m \rceil \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes \( q \text{.generate}(a + 0, a + k + 3) \) and then computes 
\[ S = \left( \sum_{j=0}^{k-1} a_j + 3 \cdot 2^{32} j \right) \mod m. \]
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 \([0, 2^w - 1]\). 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 \( \ell \).

The state transition is performed as follows:

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. 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:

1. Let \( z_1 = X_i \text{ xor } ((X_i \text{ rshift } u) \text{ bitand } d) \).
2. Let \( z_2 = z_1 \text{ xor } ((z_1 \text{ lshift } s) \text{ bitand } b) \).
3. Let \( z_3 = z_2 \text{ xor } ((z_2 \text{ lshift } t) \text{ bitand } c) \).
4. Let \( z_4 = z_3 \text{ xor } (z_3 \text{ rshift } \ell) \).

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.
/* ISO/IEC N4810 */

// 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 < m, m <= n, 2u < w, u <= w, m <= w, t <= w, l <= w, w <=
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},...,X_{i-1}, in that order.

explicit mersenne_twister_engine(result_type value);
Effects: Constructs a mersenne_twister_engine object. Sets X_{n-1} to value mod 2^w. Then, iteratively
for i = 1, ..., n-1, sets X_i to
\[ f \cdot (X_{i-1} \text{ xor } (X_{i-1} \text{ rshift } (w-2))) + i \mod n \mod 2^w. \]

Complexity: \( \Theta(n) \).
template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
Effects: Constructs a mersenne_twister_engine object. With k = [w/32] and a an array (or

equivalent) of length n k, invokes q.generate(a+0, a+n-k) and then, iteratively for i = n-1, ..., -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-1} are zero,
and if each of the other resulting X_i is 0, changes X_{n-1} 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:

(3.1) — Let \( Y = X_{i-1} - X_{i-1} - c \).
(3.2) — Set \( X_i \) to \( y = Y \mod m \). Set c to 1 if \( Y < 0 \), otherwise set c to 0.

[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 \( m^2 - m^2 + 1 \) 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:
   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_type min() { return 0; }
   static constexpr result_type_type max() { return m-1; }
   static constexpr result_type_type default_seed = 19780503u;

§ 26.6.3.3 1117
// 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 \(X_{-r},\ldots,X_{-1}\), in that order, followed by \(c\).

Effects: Constructs a subtract_with_carry_engine object. Sets the values of \(X_{-r},\ldots,X_{-1}\), in that order, as specified below. If \(X_{-1}\) 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:

\[
\text{linear_congruential_engine<result_type, 40014u,0u,2147483563u> } e(\text{value == 0u ? default_seed : value});
\]

Then, to set each \(X_k\), obtain new values \(z_0,\ldots,z_{n-1}\) from \(n = \lceil w/32 \rceil\) successive invocations of \(e\) taken modulo \(2^{32}\). Set \(X_k\) to \(\left( \sum_{j=0}^{n-1} z_j \cdot 2^{32j} \right) \mod m\).

Complexity: Exactly \(n \cdot r\) invocations of \(e\).

template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);

Effects: Constructs a subtract_with_carry_engine object. 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 \(X_{-1}\) is then 0, sets \(c\) to 1; otherwise sets \(c\) to 0.

26.6.4 Random number engine adaptors class templates

26.6.4.1 In general

Each type instantiated from a class template specified in this subclause 26.6.4 satisfies 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.
The transition algorithm discards all but \( r > 0 \) values from each block of \( p \geq r \) values delivered by \( e \). The state transition is performed as follows: If \( n \geq r \), advance the state of \( e \) from \( e_i \) to \( e_i+p-r \) and set \( n \) to 0. In any case, then increment \( n \) and advance \( e \)'s then-current state \( e_j \) to \( e_{j+1} \).

The generation algorithm yields the value returned by the last invocation of \( e() \) while advancing \( e \)'s state as described above.

```cpp
template<class Engine, size_t p, size_t r>
class discard_block_engine {
    public:
        using result_type = typename Engine::result_type;

        // engine characteristics
        static constexpr size_t block_size = p;
        static constexpr size_t used_block = r;
        static constexpr result_type min() { return Engine::min(); }
        static constexpr result_type max() { return Engine::max(); }

        // constructors and seeding functions
        discard_block_engine();
        explicit discard_block_engine(const Engine& e);
        explicit discard_block_engine(Engine&& e);
        explicit discard_block_engine(result_type s);
        template<class Sseq> explicit discard_block_engine(Sseq& q);
        void seed();
        void seed(result_type s);
        template<class Sseq> void seed(Sseq& q);

        // generating functions
        result_type operator()();
        void discard(unsigned long long z);

        // property functions
        const Engine& base() const noexcept { return e; }
    private:
        Engine e; // exposition only
        int n; // exposition only
};
```

The following relations shall hold: \( 0 < r \) and \( r \leq p \).

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:

\[ R = e \cdot \text{max}() - e \cdot \text{min}() + 1 \quad \text{and} \quad m = \lceil \log_2 R \rceil. \]

\[ w_0 = \lceil w/n \rceil, \quad n_0 = n - w \mod n, \quad y_0 = 2^{w_0} \left\lfloor R/2^{w_0} \right\rfloor, \] and \( y_1 = 2^{w_0+1} \left\lfloor R/2^{w_0+1} \right\rfloor \).

Let \( n = \lceil w/m \rceil \) if and only if the relation \( R - y_0 \leq \lfloor y_0/n \rfloor \) 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 ($k = 0; k \neq n_0; k += 1$) {
  do $u = e() - e.\min();$ while ($u \geq y_0$);
  $S = 2^{n_0} \cdot S + u \bmod 2^{n_0};$
}

for ($k = n_0; k \neq n; k += 1$) {
  do $u = e() - e.\min();$ while ($u \geq y_1$);
  $S = 2^{n_0+1} \cdot S + u \bmod 2^{n_0+1};$
}

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 [rand.adapt.shuf]

A shuffle_order_engine random number engine adaptor produces the same random numbers that are produced by some base engine $e$, but delivers them in a different sequence. The state $x_i$ of a shuffle_order_engine engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$, an additional value $Y$ of the type delivered by $e$, and an additional sequence $V$ of $k$ values also of the type delivered by $e$. The size of the state is the size of $e_i$'s state plus $k + 1$.

The transition algorithm permutes the values produced by $e$. The state transition is performed as follows:

$$j = \lfloor \frac{k \cdot (Y - e.\min)}{e.\max - e.\min + 1} \rfloor.$$ (2.1)

Set $Y$ to $V_j$ and then set $V_j$ to $e()$. (2.2)

The generation algorithm yields the last value of $Y$ produced while advancing $e$'s state as described above.

template<class Engine, size_t k>
class shuffle_order_engine {
public:
  // types
  using result_type = typename Engine::result_type;
// engine characteristics
static constexpr size_t table_size = k;
static constexpr result_type min() { return Engine::min(); }
static constexpr result_type max() { return Engine::max(); }

// constructors and seeding functions
shuffle_order_engine();
explicit shuffle_order_engine(const Engine& e);
explicit shuffle_order_engine(Engine&& e);
explicit shuffle_order_engine(result_type s);
template<class Sseq> explicit shuffle_order_engine(Sseq& q);
void seed();
void seed(result_type s);
template<class Sseq> void seed(Sseq& q);

// generating functions
result_type operator()();
void discard(unsigned long long z);

// property functions
const Engine& base() const noexcept { return e; }

private:
Engine e;  // exposition only
result_type V[k]; // exposition only
result_type Y; // exposition only

4 The following relation shall hold: 0 < k.
5 The textual representation consists of the textual representation of e, followed by the k values of V, followed by the value of Y.
6 In addition to its behavior pursuant to subclause 26.6.2.5, 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, 16807, 0, 2147483647>;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_-rand0 shall produce the value 1043618065.

using minstd_rand =
linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type minstd_-rand shall produce the value 399268537.

using mt19937 =
mersenne_twister_engine<uint_fast32_t,
32,624,397,31,0x9908b0df,11,0xffffffff,7,0x9d2c5680,15,0xefc60000,17,
0x9d2c5680,15,0xefc60000,17,0x9d2c5680,15,0xefc60000,17>;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type mt19937 shall produce the value 4123659995.

using mt19937_64 =
mersenne_twister_engine<uint_fast64_t,
64,312,156,31,0xb5026f5a96619e9,29,
0x55555555555555555555,17,
0x71d67ffeda60000,37,
0xffffffff000000000000,43,
6364136223846793006>;

Required behavior: The 10000th consecutive invocation of a default-constructed object of type mt19937_-64 shall produce 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 shall produce 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 shall produce 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 shall produce 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 shall produce 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 shall produce 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

1 A random_device uniform random bit generator produces nondeterministic random numbers.

2 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);

3   Effects: Constructs a random_device nondeterministic uniform random bit generator object. The semantics of the token parameter and the token value used by the default constructor are implementation-defined.\textsuperscript{248}

4   Throws: A value of an implementation-defined type derived from exception if the random_device could not be initialized.

double entropy() const noexcept;

5   Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate\textsuperscript{249} for the random numbers returned by operator(), in the range min() to log_2(max() + 1).

result_type operator()();

6   Returns: A nondeterministic random value, uniformly distributed between min() and max() (inclusive). It is implementation-defined how these values are generated.

7   Throws: A value of an implementation-defined type derived from exception if a random number could not be obtained.

26.6.7 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
};

seed_seq();

1   Effects: Constructs a seed_seq object as if by default-constructing its member v.

2   Throws: Nothing.

template<class T>
    seed_seq(initializer_list<T> il);

3   Requires: T shall be an integer type.

\textsuperscript{248} The parameter is intended to allow an implementation to differentiate between different sources of randomness.

\textsuperscript{249} If a device has n states whose respective probabilities are \( P_0, \ldots, P_{n-1} \), the device entropy \( S \) is defined as

\[
S = - \sum_{i=0}^{n-1} P_i \cdot \log P_i.
\]
By way of initialization, set each element of the range to the value

Effects: Same as seed_seq(il.begin(), il.end()).

```cpp
template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);
```

Requires: InputIterator shall satisfy the Cpp17InputIterator requirements (23.3.5.2). Moreover, `iterator_traits<InputIterator>::value_type` shall denote an integer type.

Effects: Constructs a seed_seq object by the following algorithm:

```cpp
for (InputIterator s = begin; s != end; ++s)
    v.push_back((s) mod 2^32);
```

```cpp
template<class RandomAccessIterator>
void generate(RandomAccessIterator begin, RandomAccessIterator end);
```

Requires: RandomAccessIterator shall satisfy the Cpp17RandomAccessIterator requirements (23.3.5.6) and the requirements of a mutable iterator. Moreover, `iterator_traits<RandomAccessIterator>::value_type` shall denote an unsigned integer type capable of accommodating 32-bit quantities.

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

\[
t = (n \geq 623) ? 11 : (n \geq 68) ? 7 : (n \geq 39) ? 5 : (n \geq 7) ? 3 : (n-1)/2;
\]

With m as the larger of s + 1 and n, transform the elements of the range: iteratively for \( k = 0, \ldots, m-1 \), calculate values

\[
\begin{align*}
    r_1 &= 1664525 \cdot T(begin[k] \ xor \ begin[k+p] \ xor \ begin[k-1]) \\
    r_2 &= r_1 + \begin{cases} 
    s, & k = 0 \\
    k \ mod \ n + v[k-1], & 0 < k \leq s \\
    k \ mod \ n, & s < k 
\end{cases}
\end{align*}
\]

and, in order, increment begin[k + p] by 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 \( k = m, \ldots, m+n-1 \), calculate values

\[
\begin{align*}
    r_3 &= 1566083941 \cdot T(begin[k] + begin[k+p] + begin[k-1]) \\
    r_4 &= r_3 - (k \ mod \ n)
\end{align*}
\]

and, in order, update begin[k + p] by xoring it with 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.

```cpp
size_t size() const noexcept;
```

Returns: The number of 32-bit units that would be returned by a call to `param()`.

Complexity: Constant time.

```cpp
template<class OutputIterator>
void param(OutputIterator dest) const;
```

Requires: OutputIterator shall satisfy the Cpp17OutputIterator requirements (23.3.5.3). Moreover, the expression `*dest = rt` shall be valid for a value `rt` of type result_type.

Effects: Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

```cpp
copy(v.begin(), v.end(), dest);
```

Throws: What and when OutputIterator operations of dest throw.
26.6.7.2 Function template generate_canonical

Each function instantiated from the template described in this subclause 26.6.7.2 maps the result of one or more invocations of a supplied uniform random bit generator \( g \) to one member of the specified \( \text{RealType} \) such that, if the values \( g_i \) produced by \( g \) are uniformly distributed, the instantiation’s results \( t_j, 0 \leq t_j < 1 \), are distributed as uniformly as possible as specified below.

[Note: 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]

\[
\text{template<class RealType, size_t bits, class URBG>}
\]

\[
\text{RealType generate_canonical(URBG& g);} \]

3

Complexity: Exactly \( k = \max(1, \lceil b / \log_2 R \rceil) \) invocations of \( g \), where \( b \) is the lesser of \( \text{numeric_limits<RealType>::digits} \) and \( \text{bits} \), and \( R \) is the value of \( g.\text{max}() - g.\text{min}() + 1 \).

4

Effects: Invokes \( g() \) \( k \) times to obtain values \( g_0, \ldots, g_{k-1} \), respectively. Calculates a quantity

\[
S = \sum_{i=0}^{k-1} (g_i - g.\text{min}()) \cdot R^i
\]

using arithmetic of type \( \text{RealType} \).

Returns: \( S/R^k \).

Throws: What and when \( g \) throws.

26.6.8 Random number distribution class templates

26.6.8.1 In general

Each type instantiated from a class template specified in this subclause 26.6.8 satisfies the requirements of a random number distribution (26.6.2.6) type.

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.

The algorithms for producing each of the specified distributions are implementation-defined.

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

A \( \text{uniform_int_distribution} \) random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function

\[
P(i | a,b) = 1/(b - a + 1) .
\]

\[
\text{template<class IntType = int>}
\]

\[
\text{class uniform_int_distribution} \{
\text{public:}
// \text{types}
\text{using result_type = IntType;}
\text{using param_type = unspecified;}

// \text{constructors and reset functions}
\text{uniform_int_distribution() : uniform_int_distribution(0) {}}
\text{explicit uniform_int_distribution(IntType a, IntType b = numeric_limits<IntType>::max());}
\text{explicit uniform_int_distribution(const param_type& parm);}
\text{void reset();}
\}
\]

\( b \) is introduced to avoid any attempt to produce more bits of randomness than can be held in \( \text{RealType} \).


// 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_int_distribution(IntType a, IntType b = numeric_limits<IntType>::max());

2 Requires: \( a \leq b \).

3 Effects: Constructs a uniform_int_distribution object; \( a \) 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]
1 A uniform_real_distribution random number distribution produces random numbers \( x, a \leq x < b \), distributed according to the constant probability density function

\[
p(x \mid a, b) = \frac{1}{b - a}.
\]

[Note: This implies that \( p(x \mid a, b) \) is undefined when \( a = b \). — end note]

template<class RealType = double>
class uniform_real_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructors and reset functions
    uniform_real_distribution() : uniform_real_distribution(0.0) {}
    explicit uniform_real_distribution(RealType a, RealType b = 1.0);
    explicit uniform_real_distribution(const param_type & parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG & g);
    template<class URBG>
    result_type operator()(URBG & g, const param_type & parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type & parm);
    result_type min() const;
    result_type max() const;
};

\[\text{§ 26.6.8.2.2 1126}\]
explicit uniform_real_distribution(RealType a, RealType b = 1.0);

  Requires: \( a \leq b \) and \( b - a \leq \text{numeric_limits<RealType>::max}() \).

  Effects: Constructs a \texttt{uniform_real_distribution} object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

result_type a() const;

  Returns: The value of the \( a \) parameter with which the object was constructed.

template<class URBG>
result_type operator()(URBG& g);

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 \texttt{bernoulli_distribution} random number distribution produces \texttt{bool} values \( b \) distributed according to the discrete probability function

\[
P(b|p) = \begin{cases} 
  p & \text{if } b = \text{true}, \\
  1 - p & \text{if } b = \text{false}.
\end{cases}
\]

```cpp
class bernoulli_distribution {
public:
  // types
  using result_type = bool;
  using param_type = unspecified;

  // constructors and reset functions
  bernoulli_distribution() : bernoulli_distribution(0.5) {} 
  explicit bernoulli_distribution(double p);
  explicit bernoulli_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URBG>
  result_type operator()(URBG& g);
  template<class URBG>
  result_type operator()(URBG& g, const param_type& parm);

  // property functions
  double p() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};

explicit bernoulli_distribution(double p);

  Requires: \( 0 \leq p \leq 1 \).

  Effects: Constructs a \texttt{bernoulli_distribution} object; \( p \) 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 \texttt{binomial_distribution} random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i|t,p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{t-i}.
\]
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 Requires: $0 \leq p \leq 1$ and $0 \leq t$.

3 Effects: Constructs a binomial_distribution object; $t$ and $p$ correspond to the respective parameters of the distribution.

IntType t() const;

4 Returns: The value of the $t$ parameter with which the object was constructed.

double p() const;

5 Returns: The value of the $p$ parameter with which the object was constructed.

### 26.6.8.3.3 Class template geometric_distribution \[rand.dist.bern.geo\]

A geometric_distribution random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid p) = p \cdot (1 - p)^i.$$
// 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);

Requires: 0 < p < 1.
Effects: Constructs a geometric_distribution object; p corresponds to the parameter of the distribution.

double p() const;
Returns: The value of the p parameter with which the object was constructed.

26.6.8.3.4 Class template negative_binomial_distribution [rand.dist.bern.negbin]

A negative_binomial_distribution random number distribution produces random integers \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i \mid k, p) = \binom{k+i-1}{i} \cdot p^k \cdot (1 - p)^i.
\]

[Note: This implies that \( P(i \mid k, p) \) is undefined when \( p == 1 \). — end note]

template<class IntType = int>
class negative_binomial_distribution {
public:
    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);

 Requires: 0 < p \leq 1 and 0 < k.
Effects: Constructs a negative_binomial_distribution object; k and p correspond to the respective parameters of the distribution.

IntType k() const;
Returns: The value of the k parameter with which the object was constructed.
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;
    };
```

```c++
explicit poisson_distribution(double mean);
```

Requires: $0 < \text{mean}$.

Effects: Constructs a poisson_distribution object; mean corresponds to the parameter of the distribution.

Returns: The value of the mean parameter with which the object was constructed.

26.6.8.4.2 Class template exponential_distribution

An exponential_distribution random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x | \lambda) = \lambda e^{-\lambda x}.$$ 

```
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);

2 Requires: 0 < lambda.
3 Effects: Constructs an exponential_distribution object; lambda corresponds to the parameter of the distribution.

RealType lambda() const;
4 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
\[ p(x | \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^\alpha \Gamma(\alpha)} \cdot x^{\alpha-1}. \]

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;
};

§ 26.6.8.4.3 1131
explicit gamma_distribution(RealType alpha, RealType beta = 1.0);

Requires: 0 < alpha and 0 < beta.

Effects: Constructs a gamma_distribution object; 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 [rand.dist.pois.weibull]

A weibull_distribution random number distribution produces random numbers \( x \geq 0 \) distributed according to the probability density function

\[ p(x | a, b) = \frac{a}{b} \left( \frac{x}{b} \right)^{a-1} \exp \left( -\left( \frac{x}{b} \right)^a \right) . \]

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;
};

explicit weibull_distribution(RealType a, RealType b = 1.0);

Requires: 0 < a and 0 < b.

Effects: Constructs a weibull_distribution object; 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.
26.6.8.4.5 Class template extreme_value_distribution

An extreme_value_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function:

\[
p(x | a, b) = \frac{1}{b} \cdot \exp \left( \frac{a - x}{b} - \exp \left( \frac{a - x}{b} \right) \right).
\]

template<class RealType = double>
class extreme_value_distribution {
    public:
        // types
        using result_type = RealType;
        using param_type = unspecified;

        // 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);

Requires: \( 0 < b \).

Effects: Constructs an extreme_value_distribution object; \( a \) 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:

\[
p(x | \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \cdot \exp \left( - \frac{(x - \mu)^2}{2\sigma^2} \right).
\]

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;

        Require: The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.

§ 26.6.8.5.1 1133
using param_type = unspecified;

// constructors and reset functions
normal_distribution() : normal_distribution(0.0) {}
explicit normal_distribution(RealType mean, RealType stddev = 1.0);
explicit normal_distribution(const param_type& parm);
void reset();

// generating functions
template<class URBG>
result_type operator()(URBG& g);
template<class URBG>
result_type operator()(URBG& g, const param_type& parm);

// property functions
RealType mean() const;
RealType stddev() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;

explicit normal_distribution(RealType mean, RealType stddev = 1.0);

Requires: 0 < stddev.
Effects: Constructs a normal_distribution object; mean and stddev correspond to the respective parameters of the distribution.

RealType mean() const;
Returns: The value of the mean parameter with which the object was constructed.

RealType stddev() const;
Returns: The value of the stddev parameter with which the object was constructed.

26.6.8.5.2 Class template lognormal_distribution [rand.dist.norm.lognormal]
A lognormal_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x \mid m,s) = \frac{1}{sx\sqrt{2\pi}} \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right).
\]

template<class RealType = double>
class lognormal_distribution {
public:
    using result_type = RealType;
    using param_type = unspecified;

    // constructors and reset functions
    lognormal_distribution() : lognormal_distribution(0.0) {}
    explicit lognormal_distribution(RealType m, RealType s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);
// property functions
RealType m() const;
RealType s() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
}

explicit lognormal_distribution(RealType m, RealType s = 1.0);

// Requires: 0 < s.
Effects: Constructs a lognormal_distribution object; m and s 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 s() const;
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

\[
p(x | n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}.
\]

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) {}
    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;
};

explicit chi_squared_distribution(RealType n);

// Requires: 0 < n.
Effects: Constructs a chi_squared_distribution object; 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;
};
```

2. Requires: 0 < b.
3. Effects: Constructs a `cauchy_distribution` object; \( a \) and \( b \) correspond to the respective parameters of the distribution.

RealType a() const;

4. Returns: The value of the \( a \) parameter with which the object was constructed.

RealType b() const;

5. 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

\[
p(x | m, n) = \frac{\Gamma((m + n)/2)}{\Gamma(m/2) \Gamma(n/2)} \cdot (\frac{m}{n})^{m/2} \cdot x^{(m/2)-1} \cdot (1 + \frac{mx}{n})^{-(m+n)/2}.
\]

```cpp
template<class RealType = double>
class fisher_f_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;
```
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) {}
    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);
};
```


```cpp
result_type min() const;
result_type max() const;
```

```cpp
explicit student_t_distribution(RealType n);
```

2

```
Requires: 0 < n.
```

3

```
Effects: Constructs a student_t_distribution object; n corresponds to the parameter of the distribution.
```

```cpp
RealType n() const;
```

4

```
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

\[
P(i | p_0, \ldots, p_{n-1}) = p_i.
\]

2

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: \( 0 < S = w_0 + \cdots + w_{n-1} \).

```cpp
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;
};
```

3

```
Effects: Constructs a discrete_distribution object with \( n = 1 \) and \( p_0 = 1 \). [Note: Such an object will always deliver the value 0. — end note]
```

```cpp
template<class InputIterator>
 discrete_distribution(InputIterator firstW, InputIterator lastW);
```

4

```
Requires: InputIterator shall satisfy the Cpp17InputIterator requirements (23.3.5.2). Moreover, iterator_traits<InputIterator>::value_type shall denote a type that is convertible to double.
```
If firstW == lastW, let n = 1 and w₀ = 1. Otherwise, (firstW, lastW) shall form 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()).

template<class UnaryOperation>
discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.14) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax − xmin)/n shall hold.

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 w_k = fw(xmin + k·δ + δ/2) for k = 0,...,n − 1.

Complexity: The number of invocations of fw shall not exceed n.

vector<double> probabilities() const;

Returns: A vector<double> whose size member returns n and whose operator[] member returns p_k when invoked with argument k for k = 0,...,n − 1.

26.6.8.6.2 Class template piecewise_constant_distribution [rand.dist.samp.pconst]

A piecewise_constant_distribution random number distribution produces random numbers x, b₀ ≤ x < bₙ, uniformly distributed over each subinterval [bᵢ, bᵢ₊₁) according to the probability density function

\[ p(x \mid b₀, \ldots, bₙ, ρ₀, \ldots, ρₙ−1) = ρᵢ \] , for bᵢ ≤ x < bᵢ₊₁.

The n + 1 distribution parameters [bᵢ], also known as this distribution’s interval boundaries, shall satisfy the relation bᵢ < bᵢ₊₁ for i = 0,...,n − 1. Unless specified otherwise, the remaining n distribution parameters are calculated as:

\[ ρ_k = \frac{w_k}{S - (b_{k+1} - b_k)} \] 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 = w₀ + ⋯ + wₙ₋₁.

template<class RealType = double>
class piecewise_constant_distribution {

public:

// types
using result_type = RealType;
using param_type = unspecified;

// constructor and reset functions
piecewise_constant_distribution();
template<class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                InputIteratorW firstW);
template<class UnaryOperation>
piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
template<class UnaryOperation>
piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax,
                                UnaryOperation fw);
explicit piecewise_constant_distribution(const param_type& parm);
void reset();

// generating functions
template<URBG>
result_type operator()(URBG& g);
template<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_constant_distribution();

Effects: Constructs a piecewise_constant_distribution object with n = 1, ρ₀ = 1, b₀ = 0, and b₁ = 1.

template<class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                         InputIteratorW firstW);

Requires: InputIteratorB and InputIteratorW shall each satisfy the C++17InputIterator requirements (23.3.5.2). Moreover, the id-expressions iterator_traits<InputIteratorB>::value_type and iterator_traits<InputIteratorW>::value_type shall each denote a type that is convertible to double. If firstB == lastB or ++firstB == lastB, let n = 1, w₀ = 1, b₀ = 0, and b₁ = 1. Otherwise, [firstB, lastB) shall form a sequence b of length n + 1, the length of the sequence w starting from firstW shall be at least n, and any wk for k ≥ n shall be 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);

Requires: Each instance of type UnaryOperation shall be a function object (20.14) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation's sole parameter.

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 wk = fw((bₖ₊₁ + bₖ)/2) for k = 0, . . . , n − 1.

Complexity: The number of invocations of fw shall not exceed n.

template<class UnaryOperation>
piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.14) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation's sole parameter. If nw = 0, let n = 1, otherwise let n = nw. The relation 0 < δ = (xmax − xmin)/n shall hold.

Effects: Constructs a piecewise_constant_distribution object with parameters taken or calculated from the following values: Let bₖ = xmin + k·δ for k = 0, . . . , n, and wk = fw(bₖ + δ/2) for k = 0, . . . , n − 1.

Complexity: The number of invocations of fw shall not exceed n.

vector<result_type> intervals() const;

Returns: A vector<result_type> whose size member returns n + 1 and whose operator[] member returns bₖ 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 ρₖ when invoked with argument k for k = 0, . . . , n − 1.

26.6.8.6.3 Class template piecewise_linear_distribution [rand.dist.samp.plinear]

A piecewise_linear_distribution random number distribution produces random numbers x, b₀ ≤ x < bₙ, distributed over each subinterval [bₖ, bₖ₊₁) according to the probability density function

\[ p(x \mid b₀, \ldots, bₙ, \rho₀, \ldots, \rhoₙ) = ρₖ \frac{bₖ₊₁ - x}{bₖ₊₁ - bₖ} + ρₖ₊₁ \frac{x - bₖ}{bₖ₊₁ - bₖ}, \text{ for } bₖ ≤ x < bₖ₊₁. \]
The \( n + 1 \) distribution parameters \( b_i \), also known as this distribution’s interval boundaries, shall satisfy the relation \( b_i < b_{i+1} \) for \( i = 0, \ldots, n - 1 \). Unless specified otherwise, the remaining \( n + 1 \) distribution parameters are calculated as \( \rho_k = w_k / S \) for \( k = 0, \ldots, n \), in which the values \( w_k \), commonly known as the weights at boundaries, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold:

\[
0 < S = \frac{1}{2} \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k).
\]

```cpp
template<class RealType = double>
class piecewise_linear_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;
    // constructor and reset functions
    piecewise_linear_distribution();
    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);
    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;
};
```

Effects: Constructs a `piecewise_linear_distribution` object with \( n = 1 \), \( \rho_0 = \rho_1 = 1 \), \( b_0 = 0 \), and \( b_1 = 1 \).

```cpp
template<class InputIteratorB, class InputIteratorW>
piecewise_linear_distribution(InputIteratorB firstB, InputIteratorB lastB,
                               InputIteratorW firstW);
```

Requires: `InputIteratorB` and `InputIteratorW` shall each satisfy the `Cpp17InputIterator` requirements (23.3.5.2). Moreover, the `id-expressions` `iterator_traits<InputIteratorB>::value_type` and `iterator_traits<InputIteratorW>::value_type` shall each denote a type that is convertible to `double`. If `firstB == lastB` or `++firstB == lastB`, let \( n = 1 \), \( \rho_0 = \rho_1 = 1 \), \( b_0 = 0 \), and \( b_1 = 1 \). Otherwise, `[firstB, lastB)` shall form a sequence \( b \) of length \( n + 1 \), the length of the sequence \( w \) starting from `firstW` shall be at least \( n + 1 \), and any \( w_k \) for \( k \geq n + 1 \) shall be ignored by the distribution.

Effects: Constructs a `piecewise_linear_distribution` object with parameters as specified above.
piecewise_linear_distribution(initializer_list<RealType> bl, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.14) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: If bl.size() < 2, let n = 1, \( \rho_0 = \rho_1 = 1 \), \( b_0 = 0 \), and \( b_1 = 1 \). Otherwise, let \([bl.begin(),bl.end())\) form a sequence \( b_0, \ldots, b_n \), and let \( w_k = \text{fw}(b_k) \) for \( k = 0, \ldots, n \).

Complexity: The number of invocations of \( \text{fw} \) shall not exceed \( n + 1 \).

template<class UnaryOperation>

piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

Requires: Each instance of type UnaryOperation shall be a function object (20.14) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s sole parameter. If \( nw = 0 \), let \( n = 1 \), otherwise let \( n = nw \). The relation \( 0 < \delta = (xmax - xmin)/n \) shall hold.

Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: Let \( b_k = xmin + k \cdot \delta \) for \( k = 0, \ldots, n \), and \( w_k = \text{fw}(b_k) \) for \( k = 0, \ldots, n \).

Complexity: The number of invocations of \( \text{fw} \) shall not exceed \( n + 1 \).

vector<result_type> intervals() const;

Returns: A vector<result_type> whose size member returns \( n + 1 \) and whose operator[] member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

vector<result_type> densities() const;

Returns: A vector<result_type> whose size member returns \( n \) and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

26.6.9 Low-quality random number generation [c.math.rand]

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 (16.5.5.9). [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 [numarray]

26.7.1 Header <valarray> synopsis [valarray.syn]

#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; // a BLAS-like slice out of an 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 indirected array

template<class T> void swap(valarray<T>&, valarray<T>&) noexcept;
}
template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator* (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> operator* (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> operator/ (const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> operator% (const typename valarray<T>::value_type&, const valarray<T>&);

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<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> operator<<(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> operator>>(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator&&(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator||(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator==(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator==(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator==(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator!=(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator!=(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator<(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator<(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator<(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator>(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator>(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator>(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);

template<class T> valarray<bool> operator<=(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<bool> operator>=(const valarray<T>&, const 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<T> abs (const valarray<T>&);

template<class T> valarray<T> acos (const valarray<T>&);

template<class T> valarray<T> asin (const valarray<T>&);

template<class T> valarray<T> atan (const valarray<T>&);

template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> atan2(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> atan2(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> cos (const valarray<T>&);

template<class T> valarray<T> cosh (const valarray<T>&);

template<class T> valarray<T> exp (const valarray<T>&);

template<class T> valarray<T> log (const valarray<T>&);

template<class T> valarray<T> log10(const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);

template<class T> valarray<T> pow(const valarray<T>&, const typename valarray<T>::value_type&);

template<class T> valarray<T> pow(const typename valarray<T>::value_type&, const valarray<T>&);

template<class T> valarray<T> sin (const valarray<T>&);

template<class T> valarray<T> sinh (const valarray<T>&);
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.

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.²⁵²

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.

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.

26.7.2 Class template `valarray`

26.7.2.1 Overview

```cpp
namespace std {
    template<class T> class valarray {
        using value_type = T;

        // 26.7.2.2, construct/destroy
        valarray();
        explicit valarray(size_t);
        valarray(const T&, size_t);
        valarray(const T*, size_t);
        valarray(valarray valarray);
        valarray(valarray valarray&&) noexcept;
        valarray(const slice_array<T>&);
        valarray(const gslice_array<T>&);
        valarray(const mask_array<T>&);
        valarray(const indirect_array<T>&);
        valarray(initializer_list<T>);
        ~valarray();

        // 26.7.2.3, assignment
        valarray& operator=(const valarray&);
        valarray& operator=(valarray valarray&&) noexcept;
        valarray& operator=(initializer_list<T>);
        valarray& operator=(const T&);
        valarray& operator=(const slice_array<T>&);
    }
}
```

²⁵² Annex B recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.
valarray& operator=(const gslice_array<T>&);
valarray& operator=(const mask_array<T>&);
valarray& operator=(const indirect_array<T>&);

// 26.7.2.4, element access
const T& operator[](size_t) const;
T& operator[](size_t);

// 26.7.2.5, subset operations
valarray operator[](slice) const;
slice_array<T> operator[](slice);
valarray operator[](const gslice& ) const;
gslice_array<T> operator[](const gslice& );
valarray operator[](const valarray<bool>&) const;
mask_array<T> operator[](const valarray<bool>&);
valarray operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);

// 26.7.2.6, unary operators
valarray operator+() const;
valarray operator-() const;
valarray operator~() const;

// 26.7.2.7, compound assignment
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 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&);

// 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());

};
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.\(^\text{253}\)

### 26.7.2.2 Constructors

#### valarray();

**Effects:** Constructs a `valarray` that has zero length.\(^\text{254}\)

#### explicit valarray(size_t n);

**Effects:** Constructs a `valarray` that has length `n`. Each element of the array is value-initialized (9.3).

#### valarray(const T& v, size_t n);

**Effects:** Constructs a `valarray` that has length `n`. Each element of the array is initialized with `v`.

**Requires:** `p` points to an array (9.2.3.4) of at least `n` elements.

**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.\(^\text{255}\)

#### valarray(const valarray&amp; 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`.\(^\text{256}\)

#### valarray(valarray&amp;&amp; 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();

**Effects:** The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

### 26.7.2.3 Assignment

#### valarray&amp; operator=(const valarray&amp; 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.

\(^\text{253}\) 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.

\(^\text{254}\) 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.

\(^\text{255}\) 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.

\(^\text{256}\) 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.
Ensures: 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>&);
Requires: 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.
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);

Requires: n < size().
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("abcdefghijklmnopqrstuvwxyz", 16);]
slice_array<T> operator[](slice slicearr);

3 Returns: An object that holds references to elements of the controlled sequence selected by slicearr.

Example:

```cpp
valarray<char> v0("abcdefgijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBghCjkDmnEp", 16);
```

— end example

valarray operator[](const gslice& gslicearr) const;

4 Returns: A valarray containing those elements of the controlled sequence designated by gslicearr.

Example:

```cpp
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<valarray<size_t> len(lv, 2), str(dv, 2));
// v0[gslice(3, len, str)] returns
// valarray<char>("dfhkmno", 6)
```

— end example

gslice_array<T> operator[](const gslice& gslicearr);

5 Returns: An object that holds references to elements of the controlled sequence selected by gslicearr.

Example:

```cpp
valarray<char> v0("abcdefgijklmnop", 16);
valarray<char> v1("ABCDEF", 6);
const size_t lv[] = { 2, 3 }; const size_t dv[] = { 7, 2 };
const valarray<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;

6 Returns: A valarray containing those elements of the controlled sequence designated by boolarr.

Example:

```cpp
const valarray<char> v0("abcdefgijklmnop", 16);
valarray<char> v1("ABC", 3);
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);

7 Returns: An object that holds references to elements of the controlled sequence selected by boolarr.

Example:

```cpp
valarray<char> v0("abcdefgijklmnop", 16);
valarray<char> v1("abcBeGhijklmnop", 16);
const bool vb[] = { false, false, true, true, false, true, true, true };  
// v0[valarray<bool>(vb, 6)] = v1;
```

— end example

valarray operator[](const valarray<size_t>& indarr) const;

8 Returns: A valarray containing those elements of the controlled sequence designated by indarr.

Example:
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t vi[] = { 7, 5, 2, 3, 8 };
// v0[valarray<size_t>(vi, 5)] returns
// valarray<char>("hfcdi", 5)
— end example

indirect_array<T> operator[](const valarray<size_t>& indarr);

9 Returns: An object that holds references to elements of the controlled sequence selected by indarr.

[Example:
valarray<char> v0("abcdefghijklmnop", 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>("abCDeBgAEjklmnop", 16)
— end example]

26.7.2.6 Unary operators

valarray operator+() const;
valarray operator-() const;
valarray operator~() const;
valarray<bool> operator!() const;

1 Requires: Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T (bool for operator!) or which may be unambiguously implicitly converted to type T (bool for operator!).

2 Returns: A valarray whose length is size(). Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

26.7.2.7 Compound assignment

valarray& operator*= (const valarray& v);
valarray& operator/= (const valarray& v);
valarray& operator%= (const valarray& v);
valarray& operator+= (const valarray& v);
valarray& operator-= (const valarray& v);
valarray& operator^= (const valarray& v);
valarray& operator&= (const valarray& v);
valarray& operator|= (const valarray& v);
valarray& operator<<=(const valarray& v);
valarray& operator>>=(const valarray& v);

1 Requires: size() == v.size(). Each of these operators may only be instantiated for a type T if the indicated operator can be applied to two operands of type T. 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.

2 Effects: Each of these operators performs the indicated operation on each of the elements of *this and the corresponding element of v.

3 Returns: *this.

4 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);
valarray& operator>>(=const T& v);

Requires: Each of these operators may only be instantiated for a type T if 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 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]

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 [valarray.nonmembers]

26.7.3.1 Binary operators [valarray.binary]

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>&);

Requires: Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which can be unambiguously implicitly converted to type T. 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.

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 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 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 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 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 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 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 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 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 typename valarray<T>::value_type&);

§ 26.7.3.1
template<class T> valarray<T> operator>>(const typename valarray<T>::value_type&, const valarray<T>&);

3 Requires: Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type T or which can be unambiguously implicitly converted to type T.

4 Returns: A valarray whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the non-array argument.

26.7.3.2 Logical operators  [valarray.comparison]

template<class T> valarray<bool> operator==(const valarray<T>&, 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 valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, 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 valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);

1 Requires: Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously implicitly converted to type bool. The two array arguments have the same length.

2 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 array arguments.

template<class T> valarray<bool> operator==(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator==(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator!=(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator< (const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator> (const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator<=(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator>=(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator&&(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<bool> operator||(const typename valarray<T>::value_type&, const valarray<T>&);

3 Requires: Each of these operators may only be instantiated for a type T to which the indicated operator can be applied and for which the indicated operator returns a value which is of type bool or which can be unambiguously implicitly converted to type 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  [valarray.transcend]

```cpp
template<class T> valarray<T> abs (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> atan2(const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<T> cos (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp (const valarray<T>&);
template<class T> valarray<T> log (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow (const valarray<T>&, const typename valarray<T>::value_type&);
template<class T> valarray<T> pow (const typename valarray<T>::value_type&, const valarray<T>&);
template<class T> valarray<T> sin (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
template<class T> valarray<T> tan (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);
```

Requires: Each of these functions may only be instantiated for a type `T` to which a unique function
with the indicated name can be applied (unqualified). This function shall return a value which is of
type `T` or which can be unambiguously implicitly converted to type `T`.

26.7.3.4 Specialized algorithms  [valarray.special]

```cpp
template<class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;
```

Effects: Equivalent to `x.swap(y)`.

26.7.4 Class slice  [class.slice]

26.7.4.1 Overview  [class.slice.overview]

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. 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.

---

257) 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.
26.7.4.3 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.

26.7.5 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
    };

}  // namespace std

This template is a helper template used by the slice subscript operator

slice_array<T> valarray<T>::operator[](slice);

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;

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;

§ 26.7.5.3
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the slice_array object refers.

26.7.5.4 Fill function [slice.arr.fill]

void operator=(const T&) const;

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the slice_array object refers.

26.7.6 The gslice class [class.gslice]

26.7.6.1 Overview [class.gslice.overview]

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;
        }
    }

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.

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_{ij} - 1.

[Example: The gslice specification

```
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:

\[
\begin{array}{cccc}
(i_0, i_1, i_2, k) = & (0, 0, 0, 3), & (0, 0, 1, 4), & (0, 0, 2, 5), & (0, 1, 0, 7), & (0, 1, 1, 8), & (0, 1, 2, 9), & (0, 2, 0, 11), & (0, 2, 1, 12), & (0, 2, 2, 13), & (0, 3, 0, 15), & (0, 3, 1, 16), & (0, 3, 2, 17), & (1, 0, 0, 22),
\end{array}
\]

§ 26.7.6.1
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.

[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

\[
\begin{align*}
(0, & 0, 0, 3), \\
(0, & 0, 1, 4), \\
(0, & 1, 0, 4), \\
(0, & 1, 1, 5), \\
(0, & 1, 2, 6), \\
\ldots
\end{align*}
\]

— end example]
5 If a degenerate slice is used as the argument to the non-const version of operator\(\[]\)(const gslice&), the behavior is undefined.

26.7.6.2 Constructors

```cpp
gslice();
gslice(size_t start, const valarray<size_t>& lengths, const valarray<size_t>& strides);
gslice(const gslice&);
```

1 The default constructor is equivalent to gslice(0, valarray<size_t>(), valarray<size_t>()). The constructor with arguments builds a gslice based on a specification of start, lengths, and strides, as explained in the previous subclause.

26.7.6.3 Access functions

```cpp
size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
```

1 Returns: The representation of the start, lengths, or strides specified for the gslice.

2 Complexity: start() is constant time. size() and stride() are linear in the number of strides.

26.7.7 Class template gslice_array

26.7.7.1 Overview

```cpp
namespace std {
    template<class T> class gslice_array {
        public:
            using value_type = T;
            void operator=(const valarray<T>&) const;
            void operator*=(const valarray<T>&) const;
            void operator/=(const valarray<T>&) const;
            void operator%=(const valarray<T>&) const;
            void operator^=(const valarray<T>&) const;
            void operator&=(const valarray<T>&) const;
            void operator|=(const valarray<T>&) const;
            void operator<<=(const valarray<T>&) const;
            void operator>>=(const valarray<T>&) const;
    };
}
```
gslice_array(const gslice_array&);
~gslice_array();
const gslice_array& operator=(const gslice_array&) const;
void operator=(const T&) const;

gslice_array() = delete; // as implied by declaring copy constructor above
};

1 This template is a helper template used by the gslice subscript operator

gslice_array<T> valarray<T>::operator[] (const gslice&);

2 It has reference semantics to a subset of an array specified by a gslice object. Thus, the expression

a[gslice(1, length, stride)] = b has the effect of assigning the elements of b to a generalized slice of the elements in a.

26.7.7.2 Assignment [gslice.array.assign]

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 [gslice.array.comp.assign]

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 [gslice.array.fill]

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 [template.mask.array]

26.7.8.1 Overview [template.mask.array.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;
};

§ 26.7.8.1 1158
This template is a helper template used by the mask subscript operator:

```
mask_array<T> valarray<T>::operator[](const valarray<bool>&).
```

It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`).

### 26.7.8.2 Assignment

```
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which 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;
```

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;
```

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;
```
indirect_array(const indirect_array&);
~indirect_array();
const indirect_array& operator=(const indirect_array&)
const;
void operator=(const Tk) 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

\texttt{a[indirect] = b;}

has the effect of assigning the elements of \texttt{b} to the elements in \texttt{a} whose indices appear in \texttt{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 \texttt{valarray<T>} object to which it refers.

2 If the \texttt{indirect_array} specifies an element in the \texttt{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 \texttt{valarray<T>} object to which the \texttt{indirect_array} object refers.

2 If the \texttt{indirect_array} specifies an element in the \texttt{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 \texttt{valarray<T>} object to which the \texttt{indirect_array} object refers.

### 26.7.10 valarray range access

In the \texttt{begin} and \texttt{end} function templates that follow, unspecified1 is a type that meets the requirements of a mutable \texttt{Cpp17RandomAccessIterator} (23.3.5.6) and models \texttt{ContiguousIterator} (23.3.4.14), whose \texttt{value_type} is the template parameter \texttt{T} and whose \texttt{reference type} is \texttt{Tk}. unspecified2 is a type that
meets the requirements of a constant \texttt{Cpp17RandomAccessIterator} and models \texttt{ContiguousIterator}, whose \texttt{value_type} is the template parameter \texttt{T} and whose \texttt{reference} type is \texttt{const T&}.

The iterators returned by \texttt{begin} and \texttt{end} for an array are guaranteed to be valid until the member function \texttt{resize(size_t, T)} (26.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

\begin{verbatim}
//template<class T> unspecified1 begin(valarray<T>& v);
//template<class T> unspecified2 begin(const valarray<T>& v); // Returns: An iterator referencing the first value in the array.

//template<class T> unspecified1 end(valarray<T>& v);
//template<class T> unspecified2 end(const valarray<T>& v); // Returns: An iterator referencing one past the last value in the array.
\end{verbatim}

26.8 Mathematical functions for floating-point types [c.math]

26.8.1 Header \texttt{<cmath>} synopsis [cmath.syn]

\begin{verbatim}
namespace std {
    using float_t = see below;
    using double_t = see below;
}

#define HUGE_VAL see below
#define HUGE_VALF see below
#define HUGE_VALL see below
#define INFINITY see below
#define NAN see below
#define FP_INFINITE see below
#define FP_NAN see below
#define FP_NORMAL see below
#define FP_SUBNORMAL see below
#define FP_ZERO see below
#define FP_FAST_FMA see below
#define FP_FAST_FMAF see below
#define FP_FAST_FMAL see below
#define FP_ILOGB0 see below
#define FP_ILOGBNAN see below
#define MATH_ERRNO see below
#define MATH_ERREXCEPT see below
#define math_errhandling see below

namespace std {
    float acos(float x); // see 16.2
    double acos(double x);
    long double acos(long double x); // see 16.2
    float acosf(float x);
    long double acosl(long double x);

    float asin(float x); // see 16.2
    double asin(double x);
    long double asin(long double x); // see 16.2
    float asinf(float x);
    long double asinl(long double x);

    float atan(float x); // see 16.2
    double atan(double x);
    long double atan(long double x); // see 16.2
    float atanf(float x);
    long double atanl(long double x);

    float atan2(float y, float x); // see 16.2
    double atan2(double y, double x);
\end{verbatim}
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);

int ilogb(float x); // see 16.2
int ilogb(double x);
int ilogb(long double x); // see 16.2
int ilogbf(float x);
int ilogbl(long double x);

float ldexp(float x, int exp); // see 16.2
double ldexp(double x, int exp);
long double ldexp(long double x, int exp); // see 16.2
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);

float log(float x); // see 16.2
double log(double x);
long double log(long double x); // see 16.2
float logf(float x);
long double logl(long double x);

float log10(float x); // see 16.2
double log10(double x);
long double log10(long double x); // see 16.2
float log10f(float x);
long double log10l(long double x);

float log1p(float x); // see 16.2
double log1p(double x);
long double log1p(long double x); // see 16.2
float log1pf(float x);
long double log1pl(long double x);

float log2(float x); // see 16.2
double log2(double x);
long double log2(long double x); // see 16.2
float log2f(float x);
long double log2l(long double x);

float logb(float x); // see 16.2
double logb(double x);
long double logb(long double x); // see 16.2
float logbf(float x);
long double logbl(long double x);

float modf(float value, float* iptr); // see 16.2
double modf(double value, double* iptr);
long double modf(long double value, long double* iptr); // see 16.2
float modff(float value, float* iptr);
long double modfl(long double value, long double* iptr);
long double modfl(long double value, long double* iptr);

float scalbn(float x, int n); // see 16.2
double scalbn(double x, int n);
long double scalbn(long double x, int n); // see 16.2
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);

float scalbin(float x, int n); // see 16.2
double scalbin(double x, long int n);
long double scalbin(long double x, long int n); // see 16.2
float scalbnf(float x, long int n);
long double scalbnl(long double x, long int n);

float cbrt(float x); // see 16.2
double cbrt(double x);
long double cbrt(long double x); // see 16.2
float cbart(float x);
long double cbartl(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 erf(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 erfcl(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);

float floor(float x); // see 16.2
double floor(double x);
long double floor(long double x); // see 16.2
float floorf(float x);
long double floorl(long double x);

float nearbyint(float x); // see 16.2
double nearbyint(double x);
long double nearbyint(long double x); // see 16.2
float nearbyintf(float x);
long double nearbyintl(long double x);

float rint(float x); // see 16.2
double rint(double x);
long double rint(long double x); // see 16.2
float rintf(float x);
long double rintl(long double x);

long int lrint(float x); // see 16.2
long int lrint(double x);
long int lrint(long double x); // see 16.2
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(float x); // see 16.2
long long int llrint(double x);
long long int llrint(long double x); // see 16.2
long long int llrintf(float x);
long long int llrintl(long double x);

float round(float x); // see 16.2
double round(double x);
long double round(long double x); // see 16.2
float roundf(float x);
long double roundl(long double x);

long int lround(float x); // see 16.2
long int lround(double x);
long int lround(long double x); // see 16.2
long int lroundf(float x);
long int lroundl(long double x);
long int lroundl(long double x);
long long int llround(float x); // see 16.2
long long int llround(double x);
long long int llroundl(long double x); // see 16.2
long long int llroundf(float x);
long double truncl(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 remquo(float x, float y, int* quo); // see 16.2
double remquo(double x, double y, int* quo);
long double remquo(long double x, long double y, int* quo); // see 16.2
float remquof(float x, float y, int* quo);
long double remquol(long double x, long double y, int* quo);

float copysign(float x, float y); // see 16.2
double copysign(double x, double y);
long double 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, float 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 fdiml(long double x, long double y);

float fmax(float x, float y); // see 16.2
double fmax(double x, double y);
long double fmax(long double x, long double y); // see 16.2
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
float fminf(float x, float y); // see 16.2
double fmin(double x, double y);
long double fminl(long double x, long double y); // see 16.2
float fminf(float x, float y);
long double fminl(long double x, long double y);
float fma(float x, float y, float z); // see 16.2
double fma(double x, double y, double z);
long double fmal(long double x, long double y, long double z); // see 16.2
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

// 26.8.4, linear interpolation
constexpr float lerp(float a, float b, float t);
constexpr double lerp(double a, double b, double t);
constexpr long double lerp(long double a, long double b, long double t);

// 26.8.5, classification / comparison functions
int fpclassify(float x);
int fpclassify(double x);
int fpclassify(long double x);
bool isfinite(float x);
bool isfinite(double x);
bool isfinite(long double x);
bool isinf(float x);
bool isinf(double x);
bool isinf(long double x);
bool isnan(float x);
bool isnan(double x);
bool isnan(long double x);
bool isnormal(float x);
bool isnormal(double x);
bool isnormal(long double x);
bool signbit(float x);
bool signbit(double x);
bool signbit(long double x);
bool isgreater(float x, float y);
bool isgreater(double x, double y);
bool isgreater(long double x, long double y);
bool isgreaterequal(float x, float y);
bool isgreaterequal(double x, double y);
bool isgreaterequal(long double x, long double y);
bool isless(float x, float y);
bool isless(double x, double y);
bool isless(long double x, long double y);
bool islessequal(float x, float y);
bool islessequal(double x, double y);
bool islessequal(long double x, long double y);
bool islessgreater(float x, float y);
bool islessgreater(double x, double y);
bool islessgreater(long double x, long double y);
bool isunordered(float x, float y);
bool isunordered(double x, double y);
bool isunordered(long double x, long double y);

// 26.8.6, mathematical special functions

// 26.8.6.1, 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);

// 26.8.6.16, Laguerre polynomials
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// 26.8.6.17, Legendre polynomials
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// 26.8.6.18, Riemann zeta function
double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double 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);

// 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);

// 26.8.6.21, spherical Neumann functions;
// spherical Bessel functions of the second kind

double sph_neumann(unsigned n, double x);
float sph_neumannf(unsigned n, float x);
long double sph_neunnl(unsigned n, long double x);

}  

1 The contents and meaning of the header `<cmath>` are the same as the C standard library header `<math.h>`, with the addition of a three-dimensional hypotenuse function (26.8.3) and the mathematical special functions described in 26.8.6. [Note: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (16.2). — end note]

2 For each set of overloaded functions within `<cmath>`, with the exception of `abs`, there shall be additional overloads sufficient to ensure:

1. If any argument of arithmetic type corresponding to a `double` parameter has type `long double`, then all arguments of arithmetic type (6.7.1) corresponding to `double` parameters are effectively cast to `long double`.

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`. 

§ 26.8.1
3. Otherwise, all arguments of arithmetic type corresponding to double parameters have type float.

[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]

    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 [c.math.hypot3]

    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 [c.math.lerp]

    constexpr float lerp(float a, float b, float t);
    constexpr double lerp(double a, double b, double t);
    constexpr long double lerp(long double a, long double b, long double t);

1 Returns: \( a + t(b - a) \).

2 Remarks: Let \( r \) be the value returned. If isfinite(a) && isfinite(b), then:

\[
\begin{align*}
(2.1) & \quad \text{If } t == 0, \text{ then } r == a. \\
(2.2) & \quad \text{If } t == 1, \text{ then } r == b. \\
(2.3) & \quad \text{If } t >= 0 \land t <= 1, \text{ then } isfinite(r). \\
(2.4) & \quad \text{If } isfinite(t) \land a == b, \text{ then } r == a. \\
(2.5) & \quad \text{If } isfinite(t) \land !isnan(t) \land b-a != 0, \text{ then } !isnan(r).
\end{align*}
\]

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 [c.math.fpclass]

1 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 [sf.cmath]

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:

\[
\begin{align*}
(1.1) & \quad \text{the function description's Returns: clause explicitly specifies a domain and those argument values fall outside the specified domain, or}
\end{align*}
\]
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);
long double assoc_laguerrel(unsigned n, unsigned m, long double x);

1 Effects: These functions compute the associated Laguerre polynomials of their respective arguments n, m, and x.

2 Returns:

\[ L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x) , \text{ for } x \geq 0, \]

where \( n \) is \( n \), \( m \) is \( m \), and \( x \) is \( x \).

3 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

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);

1 Effects: These functions compute the associated Legendre functions of their respective arguments \( l \), \( m \), and \( x \).

2 Returns:

\[ P^m_l(x) = (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_l(x) , \text{ for } |x| \leq 1, \]

where \( l \) is \( l \), \( m \) is \( m \), and \( x \) is \( x \).

3 Remarks: The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).

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);

1 Effects: These functions compute the beta function of their respective arguments \( x \) and \( y \).

2 Returns:

\[ B(x, y) = \frac{\Gamma(x) \Gamma(y)}{\Gamma(x + y)} , \text{ for } x > 0, y > 0, \]

where \( x \) is \( x \) and \( y \) is \( y \).

26.8.6.4 Complete elliptic integral of the first kind

double comp_ellint_1(double k);
float comp_ellint_1f(float k);
long double comp_ellint_1lf(long double k);

1 Effects: These functions compute the complete elliptic integral of the first kind of their respective arguments \( k \).

2 Returns:

\[ K(k) = F(k, \pi/2) , \text{ for } |k| \leq 1, \]

where \( k \) is \( k \).

3 See also 26.8.6.11.

---

258 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.
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);

1 Effects: These functions compute the complete elliptic integral of the second kind of their respective arguments \( k \).

2 Returns:

\[ E(k) = E(k, \pi/2) \text{, for } |k| \leq 1, \]

where \( k \) is \( k \).

3 See also 26.8.6.12.

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);

1 Effects: These functions compute the complete elliptic integral of the third kind of their respective arguments \( k \) and \( \nu \).

2 Returns:

\[ \Pi(\nu, k) = \Pi(\nu, k, \pi/2) \text{, for } |k| \leq 1, \]

where \( k \) is \( k \) and \( \nu \) is \( \nu \).

3 See also 26.8.6.13.

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);

1 Effects: These functions compute the regular modified cylindrical Bessel functions of their respective arguments \( \nu \) and \( x \).

2 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 \( x \).

3 Remarks: The effect of calling each of these functions is implementation-defined if \( \nu \geq 128 \).

4 See also 26.8.6.8.

26.8.6.8 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);

1 Effects: These functions compute the cylindrical Bessel functions of the first kind of their respective arguments \( \nu \) and \( x \).

2 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 \( x \).

3 Remarks: The effect of calling each of these functions is implementation-defined if \( \nu \geq 128 \).
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);

Effects: These functions compute the irregular modified cylindrical Bessel functions of their respective arguments \( \nu \) and \( x \).

Returns:

\[
K_\nu(x) = (\pi/2)i^{\nu+1}(J_\nu(ix) + iN_\nu(ix)) = \begin{cases} 
\pi \frac{1-\nu(x) - l_\nu(x)}{2 \sin \nu \pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\
\frac{\nu}{2} \lim_{\mu \to \nu} \frac{1-\mu(x) - l_\mu(x)}{\sin \mu \pi}, & \text{for } x \geq 0 \text{ and integral } \nu 
\end{cases}
\]

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 \( \nu \geq 128 \).

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

```c
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

```c
double expint(double x);
float expintf(float x);
long double expintl(long double x);
```

**Effects:** These functions compute the exponential integral of their respective arguments \( x \).

**Returns:**

\[
\text{Ei}(x) = -\int_{-x}^{\infty} e^{-t} \frac{dt}{t}
\]

where \( x \) is \( x \).

26.8.6.15 Hermite polynomials

```c
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
```

**Effects:** These functions compute the Hermite polynomials of their respective arguments \( n \) and \( x \).

**Returns:**

\[
H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}
\]

where \( n \) is \( n \) and \( x \) is \( x \).

**Remarks:** The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

26.8.6.16 Laguerre polynomials

```c
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);
```

**Effects:** These functions compute the Laguerre polynomials of their respective arguments \( n \) and \( x \).

**Returns:**

\[
L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x})^k, \text{ for } x \geq 0,
\]

where \( n \) is \( n \) and \( x \) is \( x \).

**Remarks:** The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

26.8.6.17 Legendre polynomials

```c
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);
```

**Effects:** These functions compute the Legendre polynomials of their respective arguments \( l \) and \( x \).

**Returns:**

\[
P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l, \text{ for } |x| \leq 1,
\]

where \( l \) is \( l \) and \( x \) is \( x \).

**Remarks:** The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).
26.8.6.18 Riemann zeta function

\begin{verbatim}
double riemann_zeta(double x);
float  riemann_zetaf(float x);
long double riemann_zetal(long double x);
\end{verbatim}

**Effects:** These functions compute the Riemann zeta function of their respective arguments \( x \).

\[ \zeta(x) = \begin{cases} 
\sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\
\frac{1}{1 - 2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \leq x \leq 1 \\
2^x \pi^{x-1} \sin(\frac{\pi x}{2}) \Gamma(1-x) \zeta(1-x), & \text{for } x < 0
\end{cases} \]

where \( x \) is \( x \).

26.8.6.19 Spherical Bessel functions of the first kind

\begin{verbatim}
double sph_bessel(unsigned n, double x);
float  sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);
\end{verbatim}

**Effects:** These functions compute the spherical Bessel functions of the first kind of their respective arguments \( n \) and \( x \).

\[ j_n(x) = (\pi/2x)^{1/2} J_{n+1/2}(x), \text{ for } x \geq 0, \]

where \( n \) is \( n \) and \( x \) is \( x \).

**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

\begin{verbatim}
double sph_legendre(unsigned l, unsigned m, double theta);
float  sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m, long double theta);
\end{verbatim}

**Effects:** These functions compute the spherical associated Legendre functions of their respective arguments \( l \), \( m \), and \( \theta \) (\( \theta \) measured in radians).

\[ Y^m_l(\theta, \phi) = (-1)^m \left[ \frac{(2\ell + 1)(\ell - m)!}{4\pi \ell!(\ell + m)!} \right]^{1/2} P^m_\ell(\cos \theta) e^{im\phi}, \text{ for } |m| \leq \ell, \]

where \( \ell \) is \( l \), \( m \) is \( m \), and \( \theta \) is \( \theta \).

**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

\begin{verbatim}
double sph_neumann(unsigned n, double x);
float  sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);
\end{verbatim}

**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 \( x \).

\[ n_n(x) = (\pi/2x)^{1/2} N_{n+1/2}(x), \text{ for } x \geq 0, \]
where \( n \) is \( n \) and \( x \) is \( x \).

3 Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

4 See also 26.8.6.10.
27 Time library

27.1 General

This Clause describes the chrono library (27.2) and various C functions (27.13) that provide generally useful time utilities, as summarized in Table 85.

Table 85 — Time library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.3</td>
<td>Cpp17Clock requirements</td>
</tr>
<tr>
<td>27.4</td>
<td>Time-related traits</td>
</tr>
<tr>
<td>27.5</td>
<td>Class template duration</td>
</tr>
<tr>
<td>27.6</td>
<td>Class template time_point</td>
</tr>
<tr>
<td>27.7</td>
<td>Clocks</td>
</tr>
<tr>
<td>27.8</td>
<td>Civil calendar</td>
</tr>
<tr>
<td>27.9</td>
<td>Class template time_of_day</td>
</tr>
<tr>
<td>27.10</td>
<td>Time zones</td>
</tr>
<tr>
<td>27.11</td>
<td>Formatting</td>
</tr>
<tr>
<td>27.12</td>
<td>Parsing</td>
</tr>
<tr>
<td>27.13</td>
<td>C library time utilities</td>
</tr>
</tbody>
</table>

27.2 Header <chrono> synopsis

namespace std {
    namespace chrono {
        // 27.5, class template duration
        template<class Rep, class Period = ratio<1>> class duration;
        // 27.6, class template time_point
        template<class Clock, class Duration = typename Clock::duration> class time_point;
    }
    // 27.4.3, common_type specializations
    template<class Rep1, class Period1, class Rep2, class Period2>
    struct common_type<chrono::duration<Rep1, Period1>,
        chrono::duration<Rep2, Period2>>;
    template<class Clock, class Duration1, class Duration2>
    struct common_type<chrono::time_point<Clock, Duration1>,
        chrono::time_point<Clock, Duration2>>;

    namespace chrono {
        // 27.4, customization traits
        template<class Rep> struct treat_as_floating_point;
        template<class Rep> struct duration_values;
        template<class Rep>
            inline constexpr bool treat_as_floating_point_v = treat_as_floating_point<Rep>::value;
        template<class T> struct is_clock;
        template<class T> inline constexpr bool is_clock_v = is_clock<T>::value;
        // 27.5.5, duration arithmetic
        template<class Rep1, class Period1, class Rep2, class Period2>
            constexpr common_type_t<duration<Rep1, Period1>,
                duration<Rep2, Period2>>
                operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
    }

§ 27.2
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 Period1, class Rep2, class Period>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period>& rhs);

// 27.5.6, duration comparisons

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>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// 27.5.7, duration_cast

template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);

template<class ToDuration, class Rep, class Period>
constexpr ToDuration floor(const duration<Rep, Period>& d);

template<class ToDuration, class Rep, class Period>
constexpr ToDuration ceil(const duration<Rep, Period>& d);

template<class ToDuration, class Rep, class Period, class Alloc = allocator<charT>>
constexpr ToDuration round(const duration<Rep, Period>& d);

// 27.5.10, duration I/O

template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const duration<Rep, Period>& d);

template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const duration<Rep, Period>& d);

template<class charT, class traits, class Rep, class Period, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
duration<Rep, Period>& d,
    basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// convenience typedefs
using nanoseconds = duration<
signed integer type of at least 64 bits, nano>;
using microseconds = duration<
signed integer type of at least 55 bits, micro>;
using milliseconds = duration<
signed integer type of at least 45 bits, milli>;
using seconds = duration<
signed integer type of at least 35 bits>;
using minutes = duration<
signed integer type of at least 29 bits, ratio< 60>>;
using hours = duration<
signed integer type of at least 23 bits, ratio<3600>>;
using days = duration<
signed integer type of at least 25 bits,
    ratio_multiply<ratio<24>, hours::period>>;
using weeks = duration<
signed integer type of at least 22 bits,
    ratio_multiply<ratio<7>, days::period>>;
using years = duration<
signed integer type of at least 17 bits,
    ratio_multiply<ratio<146097, 400>, days::period>>;
using months = duration<
signed integer type of at least 20 bits,
    ratio_divide<years::period, ratio<12>>>;

// 27.6.5, 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, class Duration2>
    constexpr bool operator>=(const time_point<Clock, Duration1>& lhs,
        const time_point<Clock, Duration2>& rhs);

// 27.6.7, time_point_cast
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;

template<class Duration>
  using sys_time = time_point<system_clock, Duration>;
using sys_seconds = sys_time<seconds>;
using sys_days = sys_time<days>;

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);

// 27.7.2, class utc_clock
class utc_clock;

template<class Duration>
  using utc_time = time_point<utc_clock, Duration>;
using utc_seconds = utc_time<seconds>;

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const utc_time<Duration>& t);

// 27.7.3, class tai_clock
class tai_clock;

template<class Duration>
  using tai_time = time_point<tai_clock, Duration>;
using tai_seconds = tai_time<seconds>;

§ 27.2 1180
template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const tai_time<Duration>& t);

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
  to_stream(basic_ostream<charT, traits>& os, const charT* fmt,
            const tai_time<Duration>& tp);

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>;

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const gps_time<Duration>& t);

template<class charT, class traits, class Duration>
  basic_ostream<charT, traits>&
  to_stream(basic_ostream<charT, traits>& os, const charT* fmt,
            const gps_time<Duration>& tp);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
  from_stream(basic_istream<charT, traits>& is, const charT* fmt,
              gps_time<Duration>& tp,
              basic_string<charT, traits, Alloc>* abbrev = nullptr,
              minutes* offset = nullptr);

// 27.7.5, 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);

template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
  basic_istream<charT, traits>&
  from_stream(basic_istream<charT, traits>& is, const charT* fmt,
              file_time<Duration>& tp,
              basic_string<charT, traits, Alloc>* abbrev = nullptr,
              minutes* offset = nullptr);

// 27.7.6, 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);

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt,
const local_time<Duration>& tp,
const string* abbrev = nullptr,
const seconds* offset_sec = nullptr);

template<class charT, class traits, class Duration>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
local_time<Duration>& tp,
basic_string<charT, traits, allocator<charT>>* abbrev = nullptr,
minutes* offset = nullptr);

// 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 bool operator!=(const day& x, const day& y) noexcept;

constexpr bool operator< (const day& x, const day& y) noexcept;

constexpr bool operator> (const day& x, const day& y) noexcept;

constexpr bool operator<=(const day& x, const day& y) noexcept;

constexpr bool 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>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, 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 bool operator!=(const month& x, const month& y) noexcept;

constexpr bool operator< (const month& x, const month& y) noexcept;

constexpr bool operator> (const month& x, const month& y) noexcept;

constexpr bool operator<=(const month& x, const month& y) noexcept;

constexpr bool 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>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, 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,
months* offset = nullptr);
constexpr bool operator<=(const month& x, const month& y) noexcept;
constexpr bool 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);

// 27.8.5, class year
class year;

constexpr bool operator==(const year& x, const year& y) noexcept;
constexpr bool operator!=(const year& x, const year& y) noexcept;
constexpr bool operator<(const year& x, const year& y) noexcept;
constexpr bool operator>(const year& x, const year& y) noexcept;
constexpr bool operator<=(const year& x, const year& y) noexcept;
constexpr bool 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);

// 27.8.6, class weekday
class weekday;

constexpr bool operator==(const weekday& x, const weekday& y) noexcept;
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>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, 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;
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;
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 bool operator!=(const month_day& x, const month_day& y) noexcept;
constexpr bool operator< (const month_day& x, const month_day& y) noexcept;
constexpr bool operator> (const month_day& x, const month_day& y) noexcept;
constexpr bool operator<=(const month_day& x, const month_day& y) noexcept;
constexpr bool 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>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, 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 bool operator!=(const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator< (const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator> (const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator<=(const month_day_last& x, const month_day_last& y) noexcept;
constexpr bool operator>=(const month_day_last& x, const month_day_last& y) noexcept;
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);

// 27.8.11, class month_weekday
class month_weekday;

constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;
constexpr bool operator!=(const month_weekday& x, const month_weekday& y) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_weekday& mwd);

// 27.8.12, class month_weekday_last
class month_weekday_last;

constexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;
constexpr bool operator!=(const month_weekday_last& x, const month_weekday_last& y) noexcept;

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_weekday_last& mwdl);

// 27.8.13, class year_month
class year_month;

constexpr bool operator==(const year_month& x, const year_month& y) noexcept;
constexpr bool operator!=(const year_month& x, const year_month& y) noexcept;
constexpr bool operator<(const year_month& x, const year_month& y) noexcept;
constexpr bool operator>(const year_month& x, const year_month& y) noexcept;
constexpr bool operator<=(const year_month& x, const year_month& y) noexcept;
constexpr bool 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);

template<class charT, class traits>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const year_month& ym);

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);

// 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 bool operator!=(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator<(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator>(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool operator<=(const year_month_day& x, const year_month_day& y) noexcept;
constexpr bool 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>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt,
    const year_month_day& ymd);

template<class charT, class traits, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
    year_month_day& ymd,
    basic_string<charT, traits, Alloc>* abbrev = nullptr,
    minutes* offset = nullptr);

// 27.8.15, class year_month_day_last
class year_month_day_last;

constexpr bool operator==(const year_month_day_last& x,
    const year_month_day_last& y) noexcept;
constexpr bool operator!=(const year_month_day_last& x,
    const year_month_day_last& y) noexcept;
constexpr bool operator< (const year_month_day_last& x,
    const year_month_day_last& y) noexcept;
constexpr bool operator> (const year_month_day_last& x,
    const year_month_day_last& y) noexcept;
constexpr bool operator<=(const year_month_day_last& x,
    const year_month_day_last& y) noexcept;
constexpr bool operator>=(const year_month_day_last& x,
    const year_month_day_last& y) noexcept;

constexpr year_month_day_last
operator+(const year_month_day_last& ymdl, const months& dm) noexcept;
constexpr year_month_day_last
operator+(const months& dm, const year_month_day_last& ymdl) noexcept;
constexpr year_month_day_last
operator+(const year_month_day_last& ymdl, const years& dy) noexcept;
constexpr year_month_day_last
operator+(const years& dy, const year_month_day_last& ymdl) noexcept;
constexpr year_month_day_last
operator-(const year_month_day_last& ymdl, const months& dm) noexcept;
constexpr year_month_day_last
operator-(const year_month_day_last& ymdl, const years& dy) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);

// 27.8.16, class year_month_weekday
class year_month_weekday;

constexpr bool operator==(const year_month_weekday& x,
    const year_month_weekday& y) noexcept;
constexpr bool operator!=(const year_month_weekday& x, 
const year_month_weekday& y) noexcept;

constexpr year_month_weekday
operator+(const year_month_weekday& ymwd, const months& dm) noexcept;

constexpr year_month_weekday
operator+(const months& dm, const year_month_weekday& ymwd) noexcept;

constexpr year_month_weekday
operator+(const year_month_weekday& ymwd, const years& dy) noexcept;

constexpr year_month_weekday
operator+(const years& dy, const year_month_weekday& ymwd) noexcept;

constexpr year_month_weekday
operator-(const year_month_weekday& ymwd, const months& dm) noexcept;

constexpr year_month_weekday
operator-(const year_month_weekday& ymwd, const years& dy) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

class year_month_weekday_last
constexpr bool operator==(const year_month_weekday_last& x, 
const year_month_weekday_last& y) noexcept;

constexpr bool operator!=(const year_month_weekday_last& x, 
const year_month_weekday_last& y) noexcept;

constexpr year_month_weekday_last
operator+(const year_month_weekday_last& ymwdl, const months& dm) noexcept;

constexpr year_month_weekday_last
operator+(const months& dm, const year_month_weekday_last& ymwdl) noexcept;

constexpr year_month_weekday_last
operator+(const year_month_weekday_last& ymwdl, const years& dy) noexcept;

constexpr year_month_weekday_last
operator+(const years& dy, const year_month_weekday_last& ymwdl) noexcept;

constexpr year_month_weekday_last
operator-(const year_month_weekday_last& ymwdl, const months& dm) noexcept;

constexpr year_month_weekday_last
operator-(const year_month_weekday_last& ymwdl, const years& dy) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwdl);

// 27.8.17, class year_month_weekday_last

// 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/(const day& d, const month& m) noexcept;

constexpr month_day
operator/(const day& d, int m) noexcept;

constexpr month_day_last
operator/(const month& m, last_spec) noexcept;

constexpr month_day_last
operator/(const month& m, last_spec) noexcept;
constexpr month_day_last
    operator/(last_spec, const month& m) noexcept;
constexpr month_day_last
    operator/(last_spec, int m) noexcept;
constexpr month_weekday
    operator/(const month& m, const weekday_indexed& wdi) noexcept;
constexpr month_weekday
    operator/(int m, const weekday_indexed& wdi) noexcept;
constexpr month_weekday
    operator/(const weekday_indexed& wdi, const month& m) noexcept;
constexpr month_weekday
    operator/(const weekday_indexed& wdi, 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;
constexpr year_month_day
    operator/(const year_month& ym, const day& d) noexcept;
constexpr year_month_day
    operator/(const year_month& ym, int d) noexcept;
constexpr year_month_day
    operator/(const year& y, const month_day& md) noexcept;
constexpr year_month_day
    operator/(int y, const month_day& md) noexcept;
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_month& ym, const weekday_indexed& wdi) 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 time_of_day
template<class Duration> class time_of_day;
template<> class time_of_day<hours>;
template<> class time_of_day<minutes>;
template<> class time_of_day<seconds>;
template<class Rep, class Period> class time_of_day<duration<Rep, Period>>;

template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const time_of_day<hours>& t);

template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const time_of_day<minutes>& t);

template<class charT, class traits>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const time_of_day<seconds>& t);

template<class charT, class traits, class Rep, class Period>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const time_of_day<duration<Rep, Period>>& t);

// 27.10.2, time zone database
struct tzdb;
class tzdb_list;

// 27.10.2.3, 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();

// 27.10.2.4, remote time zone database support
const tzdb& reload_tzdb();
string remote_version();

// 27.10.3, exception classes
class nonexistent_local_time;
class ambiguous_local_time;

// 27.10.4, 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);

// 27.10.5, class time_zone
enum class choose {earliest, latest};
class time_zone;

bool operator==(const time_zone& x, const time_zone& y) noexcept;
bool operator!=(const time_zone& x, const time_zone& y) noexcept;
bool operator<(const time_zone& x, const time_zone& y) noexcept;
bool operator>(const time_zone& x, const time_zone& y) noexcept;
bool operator<=(const time_zone& x, const time_zone& y) noexcept;
bool operator>=(const time_zone& x, const time_zone& y) noexcept;
// 27.10.6, class template zoned_traits
template<class T> struct zoned_traits;

// 27.10.7, 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 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);

// 27.10.8, leap second support
class leap;

bool operator==(const leap& x, const leap& y);
bool operator!=(const leap& x, const leap& y);
bool operator< (const leap& x, const leap& y);
bool operator<= (const leap& x, const leap& y);

template<class Duration>
bool operator==(const leap& x, const sys_time<Duration>& y);

template<class Duration>
bool operator==(const sys_time<Duration>& x, const leap& y);

// 27.10.9, class link
class link;
bool operator==(const link& x, const link& y);
bool operator!=(const link& x, const link& y);
bool operator< (const link& x, const link& y);
bool operator> (const link& x, const link& y);
bool operator<=(const link& x, const link& y);
bool operator>=(const link& x, const link& y);

// 27.11, formatting
template<class charT, class Streamable>
  basic_string<charT>
  format(const charT* fmt, const Streamable& s);
template<class charT, class Streamable>
  basic_string<charT>
  format(const locale& loc, const charT* fmt, const Streamable& s);
template<class charT, class traits, class Alloc, class Streamable>
  basic_string<charT, traits, Alloc>
  format(const basic_string<charT, traits, Alloc>& fmt, const Streamable& s);

// 27.12, 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};

§ 27.2
inline constexpr month December{12};
}

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
   [time.clock.req]
1 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 satisfy the requirements in Table 86.

2 In Table 86 C1 and C2 denote clock types. t1 and t2 are values returned by C1::now() where the call returning t1 happens before (6.8.2) 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>
</tbody>
</table>
Table 86 — Cpp17Clock requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1::now()</td>
<td>C1::time_point</td>
<td>Returns a time_point object representing the current point in time.</td>
</tr>
</tbody>
</table>

3 [Note: 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 satisfies the Cpp17Clock requirements (27.3),

(4.2) the types TC::rep, TC::duration, and TC::time_point meet the Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24) 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 [time.traits]

27.4.1 treat_as_floating_point [time.traits.is.fp]

```cpp
template<class Rep> struct treat_as_floating_point : is_floating_point<Rep> { };
```

1 The duration template uses the treat_as_floating_point trait to help determine if a duration object can be converted to another duration with a different tick period. If treat_as_floating_point_v<Rep> is true, then implicit conversions are allowed among durations. Otherwise, the implicit convertibility depends on the tick periods of the durations. [Note: The intention of this trait is to indicate whether a given class behaves like a floating-point type, and thus allows division of one value by another with acceptable loss of precision. If treat_as_floating_point_v<Rep> is false, Rep will be treated as if it behaved like an integral type for the purpose of these conversions. — end note]

27.4.2 duration_values [time.traits.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;
};
```

1 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;
```

2 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]

3 Remarks: The value returned shall be the additive identity.

```cpp
static constexpr Rep min() noexcept;
```

4 Returns: numeric_limits<Rep>::lowest().

5 Remarks: The value returned shall compare less than or equal to zero().
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>;
};
```

1 The period of the duration indicated by this specialization of `common_type` shall be 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
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>>;
};
```

3 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;
```

1 `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 satisfies all of the following conditions:

1.1 — the qualified-ids `T::rep`, `T::period`, `T::duration`, and `T::time_point` are valid and each denotes a type (13.9.2),
1.2 — the expression `T::is_steady` is well-formed when treated as an unevaluated operand,
1.3 — the expression `T::now()` is well-formed when treated as an unevaluated operand.

2 The behavior of a program that adds specializations for `is_clock` is undefined.

### 27.5 Class template `duration`

```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(const duration&) = default;
duration& operator=(const duration&) = default;

// 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 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;
};

// 27.5.1 Constructors 

template<class Rep2>
constexpr explicit duration(const Rep2& r);

Remarks: This constructor shall not participate in overload resolution unless Rep2 is implicitly convertible to rep and

(1.1) treat_as_floating_point_v<rep> is true or
(1.2) treat_as_floating_point_v<Rep2> is false.

duration<int, milli> d(3);        // OK
```

duration<int, milli> d(3.5);  // error

Effects: Constructs an object of type duration.

Ensures: count() == static_cast<rep>(r).

template<class Rep2, class Period2>
constexpr duration(const duration<Rep2, Period2>& d);

Remarks: This constructor shall not participate in overload resolution unless 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);
duration<int, micro> us = ms;  // OK
duration<int, milli> ms2 = us;  // error

—end example]

Effects: Constructs an object of type duration, constructing rep_ from 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: As if by ++rep_.

Returns: *this.

constexpr duration& operator++(int);

Returns: duration(rep_++).

constexpr duration& operator--();

Effects: As if by --rep_.

Returns: *this.

constexpr duration& operator--(int);

Returns: duration(rep_--).

constexpr duration& operator+=(const duration& d);

Effects: As if by: rep_ += d.count();

Returns: *this.

constexpr duration& operator-=(const duration& d);

Effects: As if by: rep_ -= d.count();

Returns: *this.

§ 27.5.3

1196
```
constexpr duration& operator*=(const rep& rhs);
   Effects: As if by: rep_ *= rhs;
   Returns: *this.
constexpr duration& operator/=(const rep& rhs);
   Effects: As if by: rep_ /= rhs;
   Returns: *this.
constexpr duration& operator%=(const rep& rhs);
   Effects: As if by: rep_ %= rhs;
   Returns: *this.
constexpr duration& operator%=(const duration& rhs);
   Effects: As if by: 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);
   Returns: CD(CD(lhs).count() - CD(rhs).count()).

template<class Rep1, class Period1, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
   operator*(const duration<Rep1, Period1>& d, const Rep2& s);
   Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible to common_type_t<Rep1, Rep2>.
   Returns: CD(d).count() * s).

template<class Rep1, class Period1, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
   operator*(const Rep1& s, const duration<Rep2, Period1>& d);
   Remarks: This operator shall not participate in overload resolution unless Rep1 is implicitly convertible to common_type_t<Rep1, Rep2>.
   Returns: d * s.

§ 27.5.5
template<class Rep1, class Period, class Rep2>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator/(const duration<Rep1, Period>& d, const Rep2& s);

Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible to common_type_t<Rep1, Rep2> and Rep2 is not a specialization of duration.

Returns: CD(CD(d).count() / s).

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr common_type_t<Rep1, Rep2>
operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Let CD be common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>.

Returns: 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);

Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible to common_type_t<Rep1, Rep2> and Rep2 is not a specialization of duration.

Returns: CD(CD(d).count() % 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);

Returns: CD(CD(lhs).count() % CD(rhs).count()).

27.5.6 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);

Returns: CT(lhs).count() == CT(rhs).count().

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator!=(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);

Returns: !(lhs == rhs).

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);

Returns: rhs < lhs.

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);

Returns: !(rhs < lhs).

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);

Returns: !(lhs < rhs).

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>=(const duration<Rep1, Period1>& lhs,
const duration<Rep2, Period2>& rhs);

Returns: !(lhs < rhs).
27.5.7 duration_cast

template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);

Remarks: This function shall not participate in overload resolution unless ToDuration is a specialization of duration.

Returns: Let CF be ratio_divide<Period, typename ToDuration::period>, and CR be common_type<typename ToDuration::rep, Rep, intmax_t>::type.

(2.1) If CF::num == 1 and CF::den == 1, returns
   
   ToDuration(static_cast<typename ToDuration::rep>(d.count()))

(2.2) Otherwise, if CF::num != 1 and CF::den == 1, returns
   
   ToDuration(static_cast<typename ToDuration::rep>(
     static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))

(2.3) Otherwise, if CF::num == 1 and CF::den != 1, returns
   
   ToDuration(static_cast<typename ToDuration::rep>(
     static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))

(2.4) Otherwise, returns
   
   ToDuration(static_cast<typename ToDuration::rep>(
     static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))

[Note: 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]

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 integral 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.

using namespace std::chrono_literals;]
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;

constexpr chrono::hours operator"h(unsigned long long hours);
constexpr chrono::duration<unspecified, ratio<3600, 1>> operator"h(long double hours);

Returns: A duration literal representing hours hours.

constexpr chrono::minutes operator"min(unsigned long long minutes);
constexpr chrono::duration<unspecified, ratio<60, 1>> operator"min(long double minutes);

Returns: A duration literal representing minutes minutes.

constexpr chrono::seconds operator"s(unsigned long long sec);
constexpr chrono::duration<unspecified> operator"s(long double sec);

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]

constexpr chrono::milliseconds operator"ms(unsigned long long msec);
constexpr chrono::duration<unspecified, milli> operator"ms(long double msec);

Returns: A duration literal representing msec milliseconds.

constexpr chrono::microseconds operator"us(unsigned long long usec);
constexpr chrono::duration<unspecified, micro> operator"us(long double usec);

Returns: A duration literal representing usec microseconds.

constexpr chrono::nanoseconds operator"ns(unsigned long long nsec);
constexpr chrono::duration<unspecified, nano> operator"ns(long double nsec);

Returns: A duration literal representing nsec nanoseconds.

27.5.9 Algorithms [time.duration.alg]

template<class Rep, class Period>
constexpr duration<Rep, Period> abs(duration<Rep, Period> d);

Remarks: This function shall not participate in overload resolution unless numeric_limits<Rep>::is_signed is true.

Returns: If d >= d.zero(), return d, otherwise return -d.

27.5.10 I/O [time.duration.io]

template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const duration<Rep, Period>& d);

Requires: Rep is an integral type whose integer conversion rank (6.7.4) is greater than or equal to that of short, or a floating point type. charT is char or wchar_t.

Effects: Forms a basic_string<charT, traits> from d.count() using to_string if charT is char, or to_wstring if charT is wchar_t. Appends the units suffix described below to the basic_string. Inserts the resulting basic_string into os. [Note: This specification ensures that the result of this streaming operation will obey the width and alignment properties of the stream. — end note]

The units suffix depends on the type Period::type as follows:

(3.1) — If Period::type is atto, the suffix is "as".
(3.2) — Otherwise, if Period::type is femto, the suffix is "fs".
(3.3) — Otherwise, if Period::type is pico, the suffix is "ps".
(3.4) — Otherwise, if Period::type is nano, the suffix is "ns".
(3.5) — Otherwise, if Period::type is micro, the suffix is "µs" ("\u00b5\u0073").
— Otherwise, if Period::type is milli, the suffix is "ms".
— Otherwise, if Period::type is centi, the suffix is "cs".
— Otherwise, if Period::type is deci, the suffix is "ds".
— Otherwise, if Period::type is ratio<1>, the suffix is "s".
— Otherwise, if Period::type is deca, the suffix is "das".
— Otherwise, if Period::type is hecto, the suffix is "hs".
— Otherwise, if Period::type is kilo, the suffix is "ks".
— Otherwise, if Period::type is mega, the suffix is "Ms".
— Otherwise, if Period::type is giga, the suffix is "Gs".
— Otherwise, if Period::type is tera, the suffix is "Ts".
— Otherwise, if Period::type is peta, the suffix is "Ps".
— Otherwise, if Period::type is exa, the suffix is "Es".
— Otherwise, if Period::type is ratio<60>, the suffix is "min".
— Otherwise, if Period::type is ratio<3600>, the suffix is "h".
— Otherwise, if Period::type::den == 1, the suffix is "[num]s".
— Otherwise, the 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.

If Period::type is micro, but the character U+00B5 cannot be represented in the encoding used for charT, the unit suffix "us" is used instead of "µs".

Returns: os.

```cpp
template<class charT, class traits, class Rep, class Period>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const duration<Rep, Period>& d);
```

Effects: Streams d into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.

Returns: os.

```cpp
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);
```

Effects: Attempts to parse the input stream is into the duration d using the format flags given in the NTCTS fmt as specified in 27.12. 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.

## Class template time_point

defines the class 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 satisfy 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 [time.point.cons]

constexpr time_point();
1 Effects: Constructs an object of type time_point, initializing d_ with duration::zero(). Such a
time_point object represents the epoch.

constexpr explicit time_point(const duration& d);
2 Effects: Constructs an object of type time_point, initializing 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 Remarks: This constructor shall not participate in overload resolution unless Duration2 is implicitly
convertible to duration.
4 Effects: Constructs an object of type time_point, initializing d_ with t.time_since_epoch().

27.6.2 Observer [time.point.observer]

constexpr duration time_since_epoch() const;
1 Returns: d_.

27.6.3 Arithmetic [time.point.arithmetic]

constexpr time_point& operator++();
1 Effects: ++d_.
2 Returns: *this.

constexpr time_point operator++(int);
3 Returns: time_point(d_++).
constexpr time_point& operator--();

Effects: --d_.
Returns: *this.

castexpr time_point operator--(int);

Returns: time_point{d.--}.

castexpr time_point& operator+=(const duration& d);

Effects: As if by: d_ += d;
Returns: *this.

castexpr time_point& operator-=(const duration& d);

Effects: As if by: d_ -= d;
Returns: *this.

27.6.4 Special values [time.point.special]

static constexpr time_point min() noexcept;

Returns: time_point(duration::min()).

static constexpr time_point max() noexcept;

Returns: time_point(duration::max()).

27.6.5 Non-member arithmetic [time.point.nonmember]

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.

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.

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.

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 [time.point.comparisons]

template<class Clock, class Duration1, class Duration2>
castexpr bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

Returns: lhs.time_since_epoch() == rhs.time_since_epoch().

template<class Clock, class Duration1, class Duration2>
castexpr bool operator!=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

Returns: !(lhs == rhs).
template<class Clock, class Duration1, class Duration2>
constexpr bool operator<(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

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);

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);

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);

Returns: !(lhs < rhs).

27.6.7 time_point_cast

[time.point.cast]
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

Remarks: This function shall not participate in overload resolution unless 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);

Remarks: This function shall not participate in overload resolution unless 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);

Remarks: This function shall not participate in overload resolution unless 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);

Remarks: This function shall not participate in overload resolution unless 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 shall satisfy the Cpp17TrivialClock requirements (27.3) unless otherwise specified.

27.7.1 Class system_clock

[time.clock.system]

27.7.1.1 Overview

namespace std::chrono {
    class system_clock {
        public:
            using rep = see below;
            using period = ratio<uns specified, uns specified>;

§ 27.7.1.1

1204
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 (and before) 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:
  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

using system_clock::rep = unspecified;

1 Requires: system_clock::duration::min() < system_clock::duration::zero() shall be true.
[Note: This implies that rep is a signed type. —end note]

static time_t to_time_t(const time_point& t) noexcept;

2 Returns: A time_t object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time_t and time_point. It is implementation-defined whether values are rounded or truncated to the required precision.

static time_point from_time_t(time_t t) noexcept;

3 Returns: A time_point object that represents the same point in time as t when both values are restricted to the coarser of the precisions of time_t and time_point. It is implementation-defined whether values are rounded or truncated to the required precision.

27.7.1.3 Non-member functions

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
operator<<<basic_ostream<charT, traits>& os, const sys_time<Duration>& tp);

1 Remarks: This operator shall not participate in overload resolution if treat_as_floating_point_v<typename Duration::rep> is true, or if Duration{1} >= days{1}.

2 Effects:
  auto const dp = floor<days>(tp);
  os << year_month_day(dp) << ' ' << time_of_day(tp-dp);

3 Returns: os.

[Example:
  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);

5 Effects: os << year_month_day(dp).

6 Returns: os.
template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
        to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const sys_time<Duration>& tp);

Effects: Streams tp into os using the format specified by the NTCTS fmt. fmt encoding follows the
rules specified in 27.11. If %Z is used, it will be replaced with "UTC" widened to charT. If %z is used
(or a modified variant of %z), an offset of 0min will be formatted.

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 27.12. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) shall be called and tp shall not be 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

27.7.2.1 Overview

namespace std::chrono {
    class utc_clock {
        public:
            using rep = a signed arithmetic type;
            using period = ratio<unspecified, unspecified>;
            using duration = chrono::duration<rep, period>;
            using time_point = chrono::time_point<utc_clock>;
            static constexpr bool is_steady = unspecified;
            static time_point now();

            template<class Duration>
                static sys_time<common_type_t<Duration, seconds>>
                    to_sys(const utc_time<Duration>& t);
            template<class Duration>
                static utc_time<common_type_t<Duration, seconds>>
                    from_sys(const sys_time<Duration>& t);
        }
    }

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. [Example:
clock_cast<utc_clock>(sys_seconds{sys_days{1970y/January/1}}).time_since_epoch() is 0s.
clock_cast<utc_clock>(sys_seconds{sys_days{2000y/January/1}}).time_since_epoch() is 946'684'822s, which is 10'957 * 86'400s + 22s.
—end example]

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

static time_point now();

Returns: from_sys(system_clock::now()), or a more accurate value of utc_time.
template<
 typename 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 leap second insertion and the last representable value of sys_time prior to the
 insertion of the leap second is returned.

 template<
 typename 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
 number 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: Calls to_stream(os, fmt, t), where fmt is a string containing "%F %T" widened to charT.

 Returns: os.

 template<class charT, class traits, class Duration>
 basic_ostream<charT, traits>&
 to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const utc_time<
 Duration>& tp);

 Effects: Streams tp into os using the format specified by the NTCTS fmt. fmt encoding follows the
 rules specified in 27.11. If %Z is used, it will be replaced with "UTC" widened to charT. If %z is used
 (or a modified variant of %z), an offset of Omin will be formatted. If tp represents a time during a leap
 second insertion, and if a seconds field is formatted, the integral portion of that format shall be "60"
 widened to charT.

 Returns: os.

 [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

 § 27.7.2.3 1207
template<class charT, class traits, class Duration, class Alloc = allocator<charT>>
basic_istream<charT, traits>&
from_stream(basic_istream<charT, traits>& is, const charT* fmt,
utc_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

Effects: Attempts to parse the input stream is into the utc_time tp using the format flags given in
the NTCTS fmt as specified in 27.12. If the parse fails to decode a valid date, is.setstate(ios_base::failbit)
shall be called and tp shall not be 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.

27.7.3 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();

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const tai_time<Duration>&) noexcept;

template<class Duration>
static tai_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration>&) noexcept;
};
}

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 falls another second behind TAI.
For example by 2000-01-01 there had been 22 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).

2 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(tai_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

§ 27.7.3.2 1208
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: Calls to_stream(os, fmt, t), where fmt is a string containing "%F %T" widened to charT.

Returns: os.

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const tai_time<Duration>& tp);

Effects: Streams tp into os using the format specified by the NTCTS fmt. fmt encoding follows the
rules specified in 27.11. If %Z is used, it will be replaced with "TAI". If %z is used (or a modified variant
of %z), an offset of 0min will be formatted. The date and time formatted shall be equivalent to that
formatted by a sys_time initialized with:
sys_time<Duration>{{tp.time_since_epoch()} -
(sys_days{1970y/January/1} - sys_days{1958y/January/1})}

Returns: os.

[Example:
auto st = sys_days{2000y/January/1};
auto tt = clock_cast<tai_clock>(st);
cout << format("%F %T %Z == ", st) << format("%F %T %Z
", 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 27.12. If the parse fails to decode a valid date, is.setstate(ios_base::failbit)
shall be called and tp shall not be 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>;
using time_point = chrono::time_point<gps_clock>;
static constexpr bool is_steady = unspecified;

static time_point now();

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const gps_time<Duration>& t) noexcept;

template<class Duration>
static gps_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration>& t) noexcept;
};

1 The clock gps_clock measures seconds since the first Sunday of January, 1980 00:00:00 UTC. Leap seconds are not inserted into GPS. Therefore every time a leap second is inserted into UTC, UTC falls another second behind GPS. Aside from the offset from 1958y/January/1 to 1980y/January/Sunday[1], GPS is behind TAI by 19s due to the 10s offset between 1958 and 1970 and the additional 9 leap seconds inserted between 1970 and 1980.

2 gps_clock is not a Cpp17TrivialClock unless the implementation can guarantee that gps_clock::now() does not propagate an exception. [Note: noexcept(from_utc(utc_clock::now())) is false. — end note]

27.7.4.2 Member functions

static time_point now();
1 Returns: from_utc(utc_clock::now()), or a more accurate value of gps_time.

template<class Duration>
static utc_time<common_type_t<Duration, seconds>>
to_utc(const gps_time<Duration>& t) noexcept;
2 Returns:
gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} + 315964809s
[Note:
315964809s == sys_days{1980y/January/Sunday[1]} - sys_days{1970y/January/1} + 9s
— end note]

template<class Duration>
static gps_time<common_type_t<Duration, seconds>>
from_utc(const utc_time<Duration>& t) noexcept;
3 Returns:
gps_time<common_type_t<Duration, seconds>>{t.time_since_epoch()} - 315964809s
[Note:
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);
1 Effects: Calls to_stream(os, fmt, t), where fmt is a string containing "%F %T" widened to charT.
2 Returns: os.
template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const gps_time<Duration>& tp);

Effects: Streams tp into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11. If %Z is used, it will be replaced with "GPS". If %z is used (or a modified variant of %z), an offset of 0min will be formatted. The date and time formatted shall be equivalent to that formatted by a sys_time initialized with:

\[
\text{sys_time<}\text{Duration}\rangle\{\text{tp.time_since_epoch()}\} + \left(\text{sys_days\{1980y/January/Sunday[1]\}} - \text{sys_days\{1970y/January/1\}}\right)
\]

Returns: os.

Example:

```cpp
auto st = sys_days{2000y/January/1};
auto gt = clock_cast<gps_clock>(st);
cout << format("%F %T %Z == ", st) << format("%F %T %Z\n", 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 27.12. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) shall be called and tp shall not be 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

```cpp
namespace std::chrono {
    using file_clock = see below;
    
    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:

```cpp
    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:

```cpp
    template<class Duration>
        static utc_time<see below>
            to_utc(const file_time<Duration>&);
```

§ 27.7.5.2 1211
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);

1 Effects: Calls to_stream(os, fmt, t), where fmt is a string containing "%F %T" widened to charT.
2 Returns: os.

template<class charT, class traits, class Duration>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const file_time<Duration>& tp);

3 Effects: Streams tp into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11. If %Z is used, it will be replaced with "UTC" widened to charT. If %z is used (or a modified variant of %z), an offset of 0min will be formatted. The date and time formatted shall be equivalent to that formatted by a sys_time initialized with clock_cast<system_clock>(tp), or by a utc_time initialized with clock_cast<utc_clock>(tp).
4 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,
file_time<Duration>& tp, basic_string<charT, traits, Alloc>* abbrev = nullptr,
minutes* offset = nullptr);

5 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 27.12. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) shall be called and tp shall not be 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.
6 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;
    };
}

1 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 = duration<rep, period>;
        using time_point = time_point<unspecified, duration>;
        static constexpr bool is_steady = unspecified;
    static time_point now() noexcept;
    };}

1 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);

2 Effects:
    os << sys_time<Duration>{lt.time_since_epoch()};

3 Returns: os.

template<class charT, class traits, class Duration>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const local_time<Duration>& tp,
    const string* abbrev = nullptr, const seconds* offset_sec = nullptr);

4 Effects: Streams tp into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11. If %Z is used, it will be replaced with *abbrev if abbrev is not equal to nullptr. If abbrev is equal to nullptr (and %Z is used), os.setstate(ios_base::failbit) shall be called. If %Z is used (or a modified variant of %Z), it will be formatted with the value of *offset_sec if offset_sec is not equal to nullptr. If %Z (or a modified variant of %Z) is used, and offset_sec is equal to nullptr, then os.setstate(ios_base::failbit) shall be called.

5 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);

6 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 27.12. If the parse fails to decode a valid date, is.setstate(ios_base::failbit) shall be called and tp shall not be 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 of %Z) is used and successfully parsed, that value will be assigned to *offset if offset is non-null.

7 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 {};

1 clock_time_conversion serves as a trait which can be used to specify how to convert a source time_point of
type time_point<SourceClock, Duration> to a destination time_point of type time_point<DestClock, Duration>
via a specialization: clock_time_conversion<DestClock, SourceClock>. A specialization
of clock_time_conversion<DestClock, SourceClock> shall provide a const-qualified operator() that
takes a parameter of type time_point<SourceClock, Duration> and returns a time_point<DestClock, OtherDuration>
representing an equivalent point in time. OtherDuration is a chrono::duration whose
specialization is computed from the input Duration in a manner which can vary for each clock_time_-
conversion specialization. A program may specialize clock_time_conversion if at least one of the template
parameters is a user-defined clock type.

2 Several specializations are provided by the implementation, as described in 27.7.9.2, 27.7.9.3, 27.7.9.4, and
27.7.9.5.

27.7.9.2 Identity conversions

template<typename Clock>
struct clock_time_conversion<Clock, Clock> {
    template<class Duration>
    time_point<Clock, Duration>
    operator()(const time_point<Clock, Duration>& t) const;
};

template<class Duration>
    time_point<Clock, Duration>
    operator()(const time_point<Clock, Duration>& t) const;
1 Returns: t.

1 template<>
struct clock_time_conversion<system_clock, system_clock> {
    template<class Duration>
    sys_time<Duration>
    operator()(const sys_time<Duration>& t) const;
};

template<class Duration>
    sys_time<Duration>
    operator()(const sys_time<Duration>& t) const;
2 Returns: t.

2 template<>
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;
3 Returns: t.

27.7.9.3 Conversions between system_clock and utc_clock

template<>
struct clock_time_conversion<utc_clock, system_clock> {
    template<class Duration>
    utc_time<common_type_t<Duration, seconds>>
    operator()(const sys_time<Duration>& t) const;
};
template<class Duration>
utc_time<common_type_t<Duration, seconds>>
operator()(const sys_time<Duration>& t) const;
  Returns: utc_clock::from_sys(t).

template<>
struct clock_time_conversion<system_clock, utc_clock> {
  template<class Duration>
  sys_time<common_type_t<Duration, seconds>>
  operator()(const utc_time<Duration>& t) const;
};

template<class Duration>
sys_time<common_type_t<Duration, seconds>>
operator()(const utc_time<Duration>& t) const;
  Returns: utc_clock::to_sys(t).

27.7.9.4 Conversions between system_clock and other clocks
  [time.clock.cast.sys]
template<class SourceClock>
struct clock_time_conversion<system_clock, SourceClock> {
  template<class Duration>
  auto operator()(const time_point<SourceClock, Duration>& t) const
  -> decltype(SourceClock::to_sys(t));
};

template<class Duration>
auto operator()(const time_point<SourceClock, Duration>& t) const
  -> decltype(SourceClock::to_sys(t));
  Remarks: This function does not participate in overload resolution unless SourceClock::to_sys(t) is well-formed. If SourceClock::to_sys(t) does not return sys_time<Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.
  Returns: SourceClock::to_sys(t).

template<class DestClock>
struct clock_time_conversion<DestClock, system_clock> {
  template<class Duration>
  auto operator()(const sys_time<Duration>& t) const
  -> decltype(DestClock::from_sys(t));
};

template<class Duration>
auto operator()(const sys_time<Duration>& t) const
  -> decltype(DestClock::from_sys(t));
  Remarks: This function does not participate in overload resolution unless DestClock::from_sys(t) is well-formed. If DestClock::from_sys(t) does not return time_point<DestClock, Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.
  Returns: DestClock::from_sys(t).

27.7.9.5 Conversions between utc_clock and other clocks
  [time.clock.cast.utc]
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));
  Remarks: This function does not participate in overload resolution unless SourceClock::to_utc(t) is well-formed. If SourceClock::to_utc(t) does not return utc_time<Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.

§ 27.7.9.5
Returns: SourceClock::to_utc(t).

```cpp
template<class DestClock>
struct clock_time_conversion<DestClock, utc_clock> {
  template<class Duration>
  auto operator()(const utc_time<Duration>& t) const
  -> decltype(DestClock::from_utc(t));
};
```

```cpp
auto operator()(const utc_time<Duration>& t) const
-> decltype(DestClock::from_utc(t));
```

Remarks: This function does not participate in overload resolution unless DestClock::from_utc(t) is well-formed. If DestClock::from_utc(t) does not return time_point<DestClock, Duration>, where Duration is a valid chrono::duration specialization, the program is ill-formed.

Returns: DestClock::from_utc(t).

### 27.7.9.6 Function template clock_cast

```cpp
template<class DestClock, class SourceClock, class Duration>
auto clock_cast(const time_point<SourceClock, Duration>& t);
```

Remarks: This function does not participate in overload resolution unless at least one of the following clock time conversion expressions is well-formed:

1. clock_time_conversion<DestClock, SourceClock>{}(t)
2. clock_time_conversion<DestClock, system_clock>{}(clock_time_conversion<system_clock, SourceClock>{}(t))
3. clock_time_conversion<DestClock, utc_clock>{}(clock_time_conversion<utc_clock, SourceClock>{}(t))
4. clock_time_conversion<DestClock, system_clock>{}(clock_time_conversion<system_clock, utc_clock>{}(clock_time_conversion<system_clock, SourceClock>{}(t)))
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. If, among the well-formed clock time conversion expressions from the above list, there is not a unique best expression, the `clock_cast` is ambiguous and the program is ill-formed.

Returns: The best well-formed clock time conversion expression in the above list.

### 27.8 The civil calendar

#### 27.8.1 In general

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

```cpp
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 is Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24), and participates in basic arithmetic with days objects, which represent a difference between two day objects.

day is a trivially copyable and standard-layout class type.

27.8.3.2 Member functions

constexpr explicit day(unsigned d) noexcept;  
1 Effects: Constructs an object of type day by initializing d_ with d. The value held is unspecified if d is not in the range [0, 255].

constexpr day& operator++() noexcept;  
2 Effects: ++d_.  
3 Returns: *this.

constexpr day operator++(int) noexcept;  
4 Effects: ++(*this).  
5 Returns: A copy of *this as it existed on entry to this member function.

constexpr day& operator--() noexcept;  
6 Effects: --d_.  
7 Returns: *this.

constexpr day operator--(int) noexcept;  
8 Effects: --(*this).  
9 Returns: A copy of *this as it existed on entry to this member function.

constexpr day& operator+=(const days& d) noexcept;  
10 Effects: *this = *this + d.  
11 Returns: *this.

constexpr day& operator-=(const days& d) noexcept;  
12 Effects: *this = *this - d.  
13 Returns: *this.
constexpr explicit operator unsigned() const noexcept;

constexpr bool ok() const noexcept;

Returns: \(1 \leq d_\& \& d_\leq 31\).

27.8.3.3 Non-member functions

constexpr bool operator==(const day& x, const day& y) noexcept;

Returns: \(\text{unsigned}\{x\} = \text{unsigned}\{y\}\).

constexpr bool operator<(const day& x, const day& y) noexcept;

Returns: \(\text{unsigned}\{x\} < \text{unsigned}\{y\}\).

constexpr day operator+(const day& x, const days& y) noexcept;

Returns: \(\text{day}(\text{unsigned}(x) + y\text{.count()})\).

constexpr day operator+(const days& x, const day& y) noexcept;

Returns: \(y + x\).

constexpr day operator-(const day& x, const days& y) noexcept;

Returns: \(x + -y\).

constexpr days operator-(const day& x, const day& y) noexcept;

Returns: \(\text{days}\{\text{int}(\text{unsigned}(x)) - \text{int}(\text{unsigned}(y))\}\).

template<class charT, class traits>

basic_ostream<charT, traits>&

operator<<(basic_ostream<charT, traits>& os, const day& d);

Effects: Inserts \text{format}(\text{fmt}, d)\ where \text{fmt} is "\%d" widened to charT. If \!d.ok(), appends with " is not a valid day".

Returns: \os\.

template<class charT, class traits>

basic_ostream<charT, traits>&

to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const day& d);

Effects: Streams \textit{d} into \os using the format specified by the NTCTS \textit{fmt}. \textit{fmt} encoding follows the rules specified in 27.11.

Returns: \os.

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 \textit{is} into the \textit{day} \textit{d} using the format flags given in the NTCTS \textit{fmt} as specified in 27.12. If the parse fails to decode a valid day, \texttt{is.setstate(ios\_base::failbit)} shall be called and \textit{d} shall not be modified. If \%Z is used and successfully parsed, that value will be assigned to \*\textit{abbrev} if \textit{abbrev} is non-null. If \%z (or a modified variant) is used and successfully parsed, that value will be assigned to \*\textit{offset} if \textit{offset} is non-null.

Returns: \is.

constexpr day operator"d(unsigned long long d) noexcept;

Returns: day{static\_cast<\text{unsigned}>(d)}.

27.8.4 Class month

27.8.4.1 Overview
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 is Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24), 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: Constructs an object of type month by initializing m_ with m. The value held is unspecified if m is not in the range [0, 255].

constexpr month& 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 month::operator unsigned() const noexcept;
14 Returns: m_.

§ 27.8.4.2
constexpr bool month::ok() const noexcept;

Returns: \(1 \leq m_ \&\& m_ \leq 12\).

27.8.4.3 Non-member functions

constexpr bool operator==(const month& x, const month& y) noexcept;

Returns: \(\text{unsigned}\{x\} == \text{unsigned}\{y\}\).

constexpr bool operator<(const month& x, const month& y) noexcept;

Returns: \(\text{unsigned}\{x\} < \text{unsigned}\{y\}\).

constexpr month operator+(const month& x, const months& y) noexcept;

Returns:

\[
\text{month}(\text{modulo}(\text{static\_cast<\text{long\_long}})(\text{unsigned}\{x\}) + (\text{y.\_count() \(-\) 1}), 12) + 1)
\]

where \(\text{modulo}(n, 12)\) 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 \text{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: If \(m\).ok() == true inserts format(os.getloc(), fmt, m) where fmt is "%b" widened to charT. Otherwise inserts unsigned{m} << " is not a valid month".

Returns: os.

template<class charT, class traits>

basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const month& m);

Effects: Streams m into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.

Returns: os.

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 27.12. If the parse fails to decode a valid month, is.setstate(ios_base::failbit) shall be called and m shall not be 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 [min(), max()]. It can be constructed with any int value, which will be subsequently truncated to fit into year’s unspecified internal storage. year is Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24), and participates in basic arithmetic with years objects, which represent a difference between two year objects.

2 year is a trivially copyable and standard-layout class type.

27.8.5.2 Member functions

constexpr explicit year(int y) noexcept;

1 Effects: Constructs an object of type year by initializing y_ with y. The value held is unspecified if y is not in the range [-32767, 32767].

constexpr year& operator++() noexcept;

2 Effects: ++y_.

3 Returns: *this.

constexpr year operator++(int) noexcept;

4 Effects: ++(*this).

5 Returns: A copy of *this as it existed on entry to this member function.

constexpr year& operator--() noexcept;

6 Effects: --y_.

7 Returns: *this.

constexpr year operator--(int) noexcept;

8 Effects: --(*this).

9 Returns: A copy of *this as it existed on entry to this member function.
constexpr year& operator+=(const years& y) noexcept;

    Effects: *this = *this + y.
    Returns: *this.

constexpr year& operator-=(const years& y) noexcept;

    Effects: *this = *this - y.
    Returns: *this.

constexpr year operator+() const noexcept;

    Returns: *this.

constexpr year year::operator-() const noexcept;

    Returns: year{-y_}.

constexpr bool is_leap() const noexcept;

    Returns: y_ % 4 == 0 && (y_ % 100 != 0 || y_ % 400 == 0).

constexpr explicit operator int() const noexcept;

    Returns: y_.

constexpr bool ok() const noexcept;

    Returns: min() <= y_ && y_ <= max().

static constexpr year min() noexcept;

    Returns: year{-32767}.

static constexpr year max() noexcept;

    Returns: year{32767}.

27.8.5.3 Non-member functions

constexpr bool operator==(const year& x, const year& y) noexcept;

    Returns: int{x} == int{y}.

constexpr bool operator<(const year& x, const year& y) noexcept;

    Returns: int{x} < int{y}.

constexpr year operator+(const year& x, const years& y) noexcept;

    Returns: year{int{x} + y.count()}.

constexpr year operator+(const years& x, const year& y) noexcept;

    Returns: y + x.

constexpr year operator-(const year& x, const years& y) noexcept;

    Returns: x + -y.

constexpr years operator-(const year& x, const year& y) noexcept;

    Returns: years{int{x} - int{y}}.

template<class charT, class traits>
    basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os, const year& y);

    Effects: Inserts format(fmt, y) where fmt is "%Y" widened to charT. If !y.ok(), appends with " is not a valid year".
    Returns: os.


```cpp
template<class charT, class traits>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const year& y);

Effects: Streams y into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.

Returns: os.
```

```cpp
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 27.12. If the parse fails to decode a valid year, is.setstate(ios_base::failbit) shall be called and y shall not be 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.
```

```cpp
constexpr year operator""y(unsigned long long y) noexcept;

Returns: year{static_cast<int>(y)}.
```

### 27.8.6 Class weekday

**27.8.6.1 Overview**

```cpp
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 explicit operator unsigned() const noexcept;
        constexpr bool ok() const noexcept;

        constexpr weekday_indexed operator[](unsigned index) const noexcept;
        constexpr weekday_last operator[](last_spec) const noexcept;
    };
}
```

1. `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` is `Cpp17EqualityComparable` (Table 23). [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]

2. `weekday` is a trivially copyable and standard-layout class type.
27.8.6.2 Member functions

```cpp
constexpr explicit weekday(unsigned wd) noexcept;
  Effects: Constructs an object of type weekday by initializing wd_ with wd. The value held is unspecified if wd is not in the range [0, 255].

constexpr weekday(const sys_days& dp) noexcept;
  Effects: Constructs an object of type weekday by computing what day of the week corresponds to the sys_days dp, and representing 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: Constructs an object of type weekday by computing what day of the week corresponds to the local_days dp, and representing that day of the week in wd_.
  Remarks: The value after construction 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 explicit operator unsigned() const noexcept;
  Returns: 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:* unsigned{x} == unsigned{y}.

**constexpr weekday operator+(const weekday& x, const days& y) noexcept;**

*Returns:* weekday(modulo(static_cast<long long>(unsigned{x}) + 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.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.ok() == true and y.ok() == true, returns a value d in the range [days{0}, 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:* If wd.ok() == true inserts format(os.getloc(), fmt, wd) where fmt is "%a" widened to charT. Otherwise inserts unsigned{wd} << " is not a valid weekday".

**Returns:** os.

**template<class charT, class traits> basic_ostream<charT, traits>& to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const weekday& wd);**

*Effects:* Streams wd into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.

*Returns:* os.

**template<class charT, class traits, class Alloc = allocator<charT>> basic_istream<charT, traits>& from_stream(basic_istream<charT, traits>& is, const charT* fmt, const 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 27.12. If the parse fails to decode a valid weekday, is.setstate(ios_base::failbit) shall be called and wd shall not be 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.

27.8.7 Class weekday_indexed

27.8.7.1 Overview

```cpp
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);
    — end example]

weekday_indexed is a trivially copyable and standard-layout class type.

27.8.7.2 Member functions [time.cal.wdidx.members]

    constexpr weekday_indexed(const chrono::weekday& wd, unsigned index) noexcept;
    Effects: Constructs an object of type weekday_indexed by initializing wd_ with wd and index_ with index. The values held are unspecified if !wd.ok() or index is not in the range [1, 5].

    constexpr chrono::weekday weekday() const noexcept;
    Returns: wd_.

    constexpr unsigned index() const noexcept;
    Returns: index_.

    constexpr bool ok() const noexcept;
    Returns: wd_.ok() && 1 <= index_ && index_ <= 5.

27.8.7.3 Non-member functions [time.cal.wdidx.nonmembers]

    constexpr bool operator==(const weekday_indexed& x, const weekday_indexed& y) noexcept;
    Returns: x.weekday() == y.weekday() && x.index() == y.index().

    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const weekday_indexed& wdi);
    Effects: os << wdi.weekday() << ']' << wdi.index(). If wdi.index() is in the range [1, 5], appends with '\'', otherwise appends with " is not a valid index".

    Returns: os.

27.8.8 Class weekday_last [time.cal.wdlast]

27.8.8.1 Overview [time.cal.wdlast.overview]

    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;
        };
    }

§ 27.8.8.1 1226
weekday_last represents the last weekday of a month.

[Note: A weekday_last object can be constructed by indexing a weekday with last. — end note] [Example:
```cpp
constexpr auto wdl = Sunday[\text{last}];  // wdl is the last Sunday of an as yet unspecified month
static_assert(wdl.weekday() == Sunday);
```
—end example]

weekday_last is a trivially copyable and standard-layout class type.

### 27.8.8.2 Member functions

```cpp
constexpr explicit weekday_last(const chrono::weekday& wd) noexcept;
```

**Effects:** Constructs an object of type weekday_last by initializing \( wd_ \) with \( wd \).

```cpp
constexpr chrono::weekday weekday() const noexcept;
```

**Returns:** \( wd_ \).

```cpp
constexpr bool ok() const noexcept;
```

**Returns:** \( \text{wd}_\text{.ok()} \).

### 27.8.8.3 Non-member functions

```cpp
constexpr bool operator==(const weekday_last& x, const weekday_last& y) noexcept;
```

**Returns:** \( x.\text{weekday()} == y.\text{weekday()} \).

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const weekday_last& wdl);
```

**Returns:** \( \text{os} \ll \text{wdl.\text{weekday()}} \ll \text{"[last]"} \).

### 27.8.9 Class month_day

#### 27.8.9.1 Overview

```cpp
namespace std::chrono {
    class month_day {
        chrono::month m_; // exposition only
        chrono::day d_; // exposition only

        public:
            month_day() = default;
            constexpr month_day(const chrono::month& m, const chrono::day& d) noexcept;

            constexpr chrono::month month() const noexcept;
            constexpr chrono::day day() const noexcept;
            constexpr bool ok() const noexcept;
    }
}
```

`month_day` represents a specific day of a specific month, but with an unspecified year. `month_day` is Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24).

### 27.8.9.2 Member functions

```cpp
constexpr month_day(const chrono::month& m, const chrono::day& d) noexcept;
```

**Effects:** Constructs an object of type month_day by initializing \( m_ \) with \( m \), and \( d_ \) with \( d \).

```cpp
constexpr chrono::month month() const noexcept;
```

**Returns:** \( m_ \).

```cpp
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 month m_; otherwise returns false. When m_ == 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 bool operator<(const month_day& x, const month_day& y) noexcept;

Returns: If x.month() < y.month() returns true. Otherwise, if x.month() > y.month() returns false. Otherwise, returns x.day() < y.day().

template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_day& md);

Returns: os << md.month() << '/' << md.day().

template<class charT, class traits>
    basic_ostream<charT, traits>&
    to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const month_day& md);

Effects: Streams md into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.

Returns: os.

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 27.12. If the parse fails to decode a valid month_day, is.setstate(ios_base::failbit) shall be called and md shall not be 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

namespace std::chrono {
    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;
    };
}

month_day_last represents the last day of a month.

[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:

constexpr auto mdl = February/last; // mdl is the last day of February of an as yet unspecified year
static_assert(mdl.month() == February);

— end example]

month_day_last is a trivially copyable and standard-layout class type.
constexpr explicit month_day_last(const chrono::month& m) noexcept;

Effects: Constructs an object of type month_day_last by initializing m_ with m.

constexpr month month() const noexcept;
Returns: m_.

constexpr bool ok() const noexcept;
Returns: m_.ok().

constexpr bool operator==(const month_day_last& x, const month_day_last& y) noexcept;
Returns: x.month() == y.month().

constexpr bool operator<(const month_day_last& x, const month_day_last& y) noexcept;
Returns: x.month() < y.month().

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_day_last& mdl);
Returns: os << mdl.month() << "/last".

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(const chrono::month& m, const chrono::weekday_indexed& wdi) noexcept;
    constexpr chrono::month month() const noexcept;
    constexpr chrono::weekday_indexed weekday_indexed() const noexcept;
    constexpr bool ok() const noexcept;
  }
};

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.

[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]

month_weekday is a trivially copyable and standard-layout class type.

27.8.11.2 Member functions

constexpr month_weekday(const chrono::month& m, const chrono::weekday_indexed& wdi) noexcept;
Effects: Constructs an object of type month_weekday by initializing 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_.ok() && wdi_.ok().
27.8.11.3 Non-member functions

```cpp
constexpr bool operator==(const month_weekday& x, const month_weekday& y) noexcept;
```

1

**Returns:**

```
x.month() == y.month() && x.weekday_indexed() == y.weekday_indexed().
```

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const month_weekday& mwd);
```

2

**Returns:**

```
os << mwd.month() << '/' << mwd.weekday_indexed().
```

27.8.12 Class month_weekday_last

27.8.12.1 Overview

```cpp
namespace std::chrono {
    class month_weekday_last {
        chrono::month m_; // exposition only
        chrono::weekday_last wdl_; // exposition only
        public:
            constexpr month_weekday_last(const chrono::month& m,
                                          const chrono::weekday_last& wdl) noexcept;
            constexpr chrono::month month() const noexcept;
            constexpr chrono::weekday_last weekday_last() const noexcept;
            constexpr bool ok() const noexcept;
    }
}
```

1

`month_weekday_last` represents the last weekday of a month, of an as yet unspecified year. To do this the `month_weekday_last` stores a `month` and a `weekday_last`.

2

**Example:**

```cpp
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]);
```

1

`month_weekday_last` is a trivially copyable and standard-layout class type.

27.8.12.2 Member functions

```cpp
constexpr month_weekday_last(const chrono::month& m,
                             const chrono::weekday_last& wdl) noexcept;
```

1

**Effects:** Constructs an object of type `month_weekday_last` by initializing `m_` with `m`, and `wdl_` with `wdl`.

```cpp
constexpr chrono::month month() const noexcept;
```

2

**Returns:**

```
m_.
```

```cpp
constexpr chrono::weekday_last weekday_last() const noexcept;
```

3

**Returns:**

```
wdl_.
```

```cpp
constexpr bool ok() const noexcept;
```

4

**Returns:**

```
m_.ok() && wdl_.ok().
```

27.8.12.3 Non-member functions

```cpp
constexpr bool operator==(const month_weekday_last& x, const month_weekday_last& y) noexcept;
```

1

**Returns:**

```
x.month() == y.month() && x.weekday_last() == y.weekday_last().
```
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const month_weekday_last& mwdl);

    Returns: os << mwdl.month() << '/' << mwdl.weekday_last().

27.8.13 Class year_month [time.cal.ym]

27.8.13.1 Overview [time.cal.ym.overview]

namespace std::chrono {
    class year_month {
        chrono::year y_; // exposition only
        chrono::month m_; // exposition only

    public:
        year_month() = default;
        constexpr year_month(const chrono::year& y, const chrono::month& m) noexcept;
        constexpr chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr year_month& operator+=(const months& dm) noexcept;
        constexpr year_month& operator-=(const months& dm) noexcept;
        constexpr year_month& operator+=(const years& dy) noexcept;
        constexpr year_month& operator-=(const years& dy) noexcept;
        constexpr bool ok() const noexcept;
    };
}

1. 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 is Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24).

2. year_month is a trivially copyable and standard-layout class type.

27.8.13.2 Member functions [time.cal.ym.members]

constexpr year_month(const chrono::year& y, const chrono::month& m) noexcept;

1. Effects: Constructs an object of type year_month by initializing y_ with y, and m_ with m.

constexpr chrono::year year() const noexcept;

2. Returns: y_.

constexpr chrono::month month() const noexcept;

3. Returns: m_.

constexpr year_month& operator+=(const months& dm) noexcept;

4. Effects: *this = *this + dm.

5. Returns: *this.

constexpr year_month& operator-=(const months& dm) noexcept;

6. Effects: *this = *this - dm.

7. Returns: *this.

constexpr year_month& operator+=(const years& dy) noexcept;

8. Effects: *this = *this + dy.

9. Returns: *this.

constexpr year_month& operator-=(const years& dy) noexcept;

10. Effects: *this = *this - dy.

11. Returns: *this.

§ 27.8.13.2
27.8.13.3 Non-member functions

```cpp
cstexpr bool ok() const noexcept;
Returns: y_.ok() && m_.ok().
```

```cpp
cstexpr bool operator==(const year_month& x, const year_month& y) noexcept;
Returns: x.year() == y.year() && x.month() == y.month().
```

```cpp
cstexpr bool operator<(const year_month& x, const year_month& y) noexcept;
Returns: If x.year() < y.year() returns true. Otherwise, if x.year() > y.year() returns false. Otherwise, returns x.month() < y.month().
```

```cpp
cstexpr year_month operator+(const year_month& ym, const months& dm) noexcept;
Returns: A year_month value z such that z - ym == dm.
Complexity: \( \Theta(1) \) with respect to the value of dm.
```

```cpp
cstexpr year_month operator+(const months& dm, const year_month& ym) noexcept;
Returns: ym + dm.
```

```cpp
cstexpr year_month operator-(const year_month& ym, const months& dm) noexcept;
Returns: ym + -dm.
```

```cpp
cstexpr 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()})}
```

```cpp
cstexpr year_month operator+(const year_month& ym, const years& dy) noexcept;
Returns: (ym.year() + dy) / ym.month().
```

```cpp
cstexpr year_month operator+(const years& dy, const year_month& ym) noexcept;
Returns: ym + dy.
```

```cpp
cstexpr year_month operator-(const year_month& ym, const years& dy) noexcept;
Returns: ym + -dy.
```

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month& ym);
Returns: os << ym.year() << '/' << ym.month().
```

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>&
to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const year_month& ym);
Effects: Streams ym into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.
Returns: os.
```

```cpp
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 27.12. If the parse fails to decode a valid year_month, is.setstate(ios_base::failbit) shall be called and ym shall not be 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.
```

§ 27.8.13.3
27.8.14 Class `year_month_day`

### Overview

```cpp
namespace std::chrono {
    class year_month_day {
    /\ exposition only
        chrono::year y_;          // exposition only
        chrono::month m_;         // exposition only
        chrono::day d_;           // exposition only

    public:
        year_month_day() = default;
        constexpr year_month_day(const chrono::year& y, const chrono::month& m,
                                   const chrono::day& d) noexcept;
        constexpr year_month_day(const year_month_day_last& ymdl) noexcept;
        constexpr year_month_day(const sys_days& dp) noexcept;
        constexpr explicit year_month_day(const local_days& dp) noexcept;

        constexpr year_month_day& operator+=(const months& m) noexcept;
        constexpr year_month_day& operator-=(const months& m) noexcept;
        constexpr year_month_day& operator+=(const years& y) noexcept;
        constexpr year_month_day& operator-=(const years& y) noexcept;

        constexpr chrono::year year() const noexcept;
        constexpr chrono::month month() const noexcept;
        constexpr chrono::day day() const noexcept;

        constexpr operator sys_days() const noexcept;
        constexpr explicit operator local_days() const noexcept;
        constexpr bool ok() const noexcept;
    };
}
```

1. `year_month_day` represents a specific year, month, and day. `year_month_day` is a field-based time point with a resolution of `days`. [Note: `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` is `Cpp17EqualityComparable` (Table 23) and `Cpp17LessThanComparable` (Table 24).

2. `year_month_day` is a trivially copyable and standard-layout class type.

### Member functions

```cpp
constexpr year_month_day(const chrono::year& y, const chrono::month& m,
                          const chrono::day& d) noexcept;
1   Effects: Constructs an object of type `year_month_day` by initializing `y_` with `y`, `m_` with `m`, and `d_` with `d`.

constexpr year_month_day(const year_month_day_last& ymdl) noexcept;
2   Effects: Constructs an object of type `year_month_day` by initializing `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`.
4   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;
5   Effects: Constructs an object of type `year_month_day` that corresponds to the date represented by `dp`.
6   Remarks: Equivalent to constructing with `sys_days{dp.time_since_epoch()}`.
```
constexpr year_month_day& operator+=(const months& m) noexcept;

Effects: *this = *this + m.
Returns: *this.

constexpr year_month_day& operator-=(const months& m) noexcept;

Effects: *this = *this - m.
Returns: *this.

constexpr year_month_day& operator+=(const years& y) noexcept;

Effects: *this = *this + y.
Returns: *this.

constexpr year_month_day& operator-=(const years& y) noexcept;

Effects: *this = *this - y.
Returns: *this.

constexpr chrono::year year() const noexcept;
Returns: y_.

constexpr chrono::month month() const noexcept;
Returns: m_.

constexpr chrono::day day() const noexcept;
Returns: d_.

constexpr operator sys_days() const noexcept;

Returns: If ok(), returns a sys_days holding a count of days from the sys_days epoch to *this (a negative value if *this represents a date prior to the sys_days epoch). Otherwise, if y_.ok() && m_.ok() is true, returns a sys_days which is offset from sys_days{y_/m_/last}.day() by the number of days d_ is offset from sys_days{y_/m_/last}.day(). Otherwise the value returned is unspecified.

Remarks: A sys_days in the range [days{-12687428}, days{11248737}] which is converted to a year_month_day shall have the same value when converted back to a sys_days.

[Example:]
static_assert(year_month_day{sys_days{2017y/January/0}} == 2016y/December/31);
static_assert(year_month_day{sys_days{2017y/January/31}} == 2017y/January/31);
static_assert(year_month_day{sys_days{2017y/January/32}} == 2017y/February/1);
—end example]

constexpr explicit operator local_days() const noexcept;
Returns: local_days{sys_days{*this}.time_since_epoch()}.

constexpr bool ok() const noexcept;
Returns: If y_.ok() is true, and m_.ok() is true, and d_ is in the range [1d, (y_/m_/last).day()], then returns true; otherwise returns false.

27.8.14.3 Non-member functions

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 bool operator<(const year_month_day& x, const year_month_day& y) noexcept;

Returns: If x.year() < y.year(), returns true. Otherwise, if x.year() > y.year(), returns false. Otherwise, if x.month() < y.month(), returns true. Otherwise, if x.month() > y.month(), returns false. Otherwise, returns x.day() < y.day().

§ 27.8.14.3
constexpr year_month_day operator+(const year_month_day& ymd, const months& dm) noexcept;
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;
Returns: ymd + dm.

constexpr year_month_day operator-(const year_month_day& ymd, const months& dm) noexcept;
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: Inserts format(fmt, ymd) where fmt is "%F" widened to charT. If !ymd.ok(), appends with " is not a valid date".
Returns: os.

template<class charT, class traits>
basic_ostream<charT, traits>& to_stream(basic_ostream<charT, traits>& os, const charT* fmt, const year_month_day& ymd);
Effects: Streams ymd into os using the format specified by the NTCTS fmt. fmt encoding follows the rules specified in 27.11.
Returns: os.

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 27.12. If the parse fails to decode a valid year_month_day, is.setstate(ios_base::failbit) shall be called and ymd shall not be 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

namespace std::chrono {
    class year_month_day_last {
        chrono::year y_; // exposition only
        chrono::month_day_last mdl_; // exposition only
    }
}
public:
  constexpr year_month_day_last(const chrono::year& y,
      const chrono::month_day_last& mdl) noexcept;

  constexpr year_month_day_last& operator+=(const months& m) noexcept;
  constexpr year_month_day_last& operator-=(const months& m) noexcept;
  constexpr year_month_day_last& operator+=(const years& y) noexcept;
  constexpr year_month_day_last& operator-=(const years& y) noexcept;

  constexpr chrono::year year() const noexcept;
  constexpr chrono::month month() const noexcept;
  constexpr chrono::month_day_last month_day_last() const noexcept;
  constexpr chrono::day day() const noexcept;
  constexpr operator sys_days() const noexcept;
  constexpr explicit operator local_days() const noexcept;
  constexpr bool ok() const noexcept;
};

1 year_month_day_last represents the last day of a specific year and month. year_month_day_last is a field-based time point with a resolution of days, except that it is restricted to pointing to the last day of a year and month. [Note: year_month_day_last supports years- and months-oriented arithmetic, but not days-oriented arithmetic. For the latter, there is a conversion to sys_days, which efficiently supports days-oriented arithmetic. — end note] year_month_day_last is Cpp17EqualityComparable (Table 23) and Cpp17LessThanComparable (Table 24),

2 year_month_day_last is a trivially copyable and standard-layout class type.

### 27.8.15.2 Member functions

```cpp
constexpr year_month_day_last(const chrono::year& y,
      const chrono::month_day_last& mdl) noexcept;

1 Effects: Constructs an object of type year_month_day_last by initializing y_ with y and mdl_ with mdl.

constexpr year_month_day_last& operator+=(const months& m) noexcept;
2 Effects: *this = *this + m.
3 Returns: *this.

constexpr year_month_day_last& operator-=(const months& m) noexcept;
4 Effects: *this = *this - m.
5 Returns: *this.

constexpr year_month_day_last& operator+=(const years& y) noexcept;
6 Effects: *this = *this + y.
7 Returns: *this.

constexpr year_month_day_last& operator-=(const years& y) noexcept;
8 Effects: *this = *this - y.
9 Returns: *this.

constexpr chrono::year year() const noexcept;
10 Returns: y_.

constexpr chrono::month month() const noexcept;
11 Returns: mdl_.month().

constexpr chrono::month_day_last month_day_last() const noexcept;
12 Returns: mdl_.
```
constexpr chrono::day day() const noexcept;

Returns: A day representing the last day of the (year, month) pair represented by *this. [Note: This value may be computed on demand. —end note]

constexpr operator sys_days() const noexcept;

Returns: sys_days{year()/month()/day()}.[27.8.15.3 Non-member functions]

constexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.[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
    }
}

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_day_last& ymdl);

Returns: os << ymdl.year() << '/' << ymdl.month_day_last().
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 \(n^{th}\) 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 is Cpp17EqualityComparable (Table 23).

2 year_month_weekday is a trivially copyable and standard-layout class type.

27.8.16.2 Member functions

1 constexpr year_month_weekday(const chrono::year& y, const chrono::month& m,
                              const chrono::weekday_indexed& wdi) noexcept;

   Effects: Constructs an object of type year_month_weekday by initializing \(y_\) with \(y\), \(m_\) with \(m\), and \(wdi_\) with \(wdi\).

2 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.

3 constexpr explicit year_month_weekday(const local_days& dp) noexcept;

   Effects: Constructs an object of type year_month_weekday that corresponds to the date represented by \(dp\).

   Remarks: Equivalent to constructing with sys_days{dp.time_since_epoch()}.

4 constexpr year_month_weekday& operator+=(const months& m) noexcept;

   Effects: *this = *this + m.

   Returns: *this.

5 constexpr year_month_weekday& operator-=(const months& m) noexcept;

   Effects: *this = *this - m.

   Returns: *this.

6 constexpr year_month_weekday& operator+=(const years& y) noexcept;

   Effects: *this = *this + y. 

§ 27.8.16.2
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;

constexpr operator sys_days() const noexcept;

Returns: If y_.ok() && m_.ok() && wdi_.weekday().ok(), returns a sys_days that represents the date (index() - 1) * 7 days after the first weekday() of year()/month(). If index() is 0 the returned sys_days represents the date 7 days prior to the first weekday() of year()/month(). Otherwise the returned value is unspecified.

constexpr explicit operator local_days() const noexcept;

Returns: local_days{sys_days{*this}.time_since_epoch()}.  

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;

Returns: (ymwd.year() / ymwd.month() + dm) / ymwd.weekday_indexed().

constexpr year_month_weekday operator+(const months& dm, const year_month_weekday& ymwd) noexcept;

Returns: ymwd + dm.

constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const months& dm) noexcept;

Returns: ymwd + (-dm).

constexpr year_month_weekday operator+(const year_month_weekday& ymwd, const years& dy) noexcept;

Returns: {ymwd.year()+dy, ymwd.month(), ymwd.weekday_indexed()}.  

constexpr year_month_weekday operator+(const years& dy, const year_month_weekday& ymwd) noexcept;

Returns: ymwd + dm.

constexpr year_month_weekday operator-(const year_month_weekday& ymwd, const years& dy) noexcept;

Returns: ymwd + (-dm).
template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const year_month_weekday& ymwd);

8  Returns: os << ymwdi.year() << '/' << ymwdi.month() << '/' << ymwdi.weekday_indexed().

27.8.17  Class year_month_weekday_last
8
27.8.17.1  Overview
8
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 is Cpp17EqualityComparable (Table 23).

27.8.17.2  Member functions
8
1  year_month_weekday_last is a trivially copyable and standard-layout class type.

constexpr year_month_weekday_last(const chrono::year& y, const chrono::month& m,
        const chrono::weekday_last& wdl) noexcept;

2  Effects: Constructs an object of type year_month_weekday_last by initializing y_ with y, m_ with m, and wdl_ with wdl.

constexpr year_month_weekday_last& operator+=(const months& m) noexcept;

3  Effects: *this = *this + m.

4  Returns: *this.

constexpr year_month_weekday_last& operator-=(const months& m) noexcept;

5  Effects: *this = *this - m.

6  Returns: *this.

constexpr year_month_weekday_last& operator+=(const years& y) noexcept;

7  Effects: *this = *this + y.

8  Returns: *this.
```cpp
constexpr year_month_weekday_last& operator-=(const years& y) noexcept;

Effects: \*this = \*this - y.
Returns: \*this.

constexpr chrono::year() const noexcept;
Returns: y_.

constexpr chrono::month() const noexcept;
Returns: m_.

constexpr chrono::weekday() const noexcept;
Returns: wdl_.weekday().

constexpr chrono::weekday_last() const noexcept;
Returns: wdl_.

constexpr operator sys_days() const noexcept;
Returns: If ok() == true, returns a sys_days that represents the last weekday() of year()/month(). Otherwise the returned value is unspecified.

constexpr explicit operator local_days() const noexcept;
Returns: local_days{sys_days{*this}.time_since_epoch()}. 

constexpr bool ok() const noexcept;
Returns: y_.ok() && m_.ok() && wdl_.ok().

27.8.17.3 Non-member functions

constexpr bool operator==(const year_month_weekday_last& x, const year_month_weekday_last& y) noexcept;
Returns: x.year() == y.year() && x.month() == y.month() && x.weekday_last() == y.weekday_last()

constexpr year_month_weekday_last operator+(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
Returns: (ymwdl.year() / ymwdl.month() + dm) / ymwdl.weekday_last().

constexpr year_month_weekday_last operator+(const months& dm, const year_month_weekday_last& ymwdl) noexcept;
Returns: ymwdl + dm.

constexpr year_month_weekday_last operator-(const year_month_weekday_last& ymwdl, const months& dm) noexcept;
Returns: ymwdl + (-dm).

constexpr year_month_weekday_last operator+(const years& dy, const year_month_weekday_last& ymwdl) noexcept;
Returns: {ymwdl.year()+dy, ymwdl.month(), ymwdl.weekday_last()}. 

constexpr year_month_weekday_last operator+(const years& dy, const year_month_weekday_last& ymwdl) noexcept;
Returns: ymwdl + dy.

constexpr year_month_weekday_last operator-(const year_month_weekday_last& ymwdl, const years& dy) noexcept;
Returns: ymwdl + (-dy).
```

§ 27.8.17.3
template<class charT, class traits>&
operator<<(basic_ostream<charT, traits>& os, const year_month_weekday_last& ymwdl);

Returns: os << ymwdl.year() << '/' << ymwdl.month() << '/' << ymwdl.weekday_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:
year_month ym = 2015y/April;
month_day md1 = April/4;
month_day md2 = 4d/April;
—end note]

[Example:
auto a = 2015/4/4; // a == int(125)
auto b = 2015y/4/4; // b == year_month_day{year(2015), month(4), day(4)}
auto c = 2015y/4d/April; // error: no viable operator/ for first /
auto d = 2015/April/4; // error: no viable operator/ for first /
—end example]

constexpr year_month
operator/(const year& y, const month& m) noexcept;

Returns: {y, m}.

constexpr year_month
operator/(const year& y, int m) noexcept;

Returns: y / month(m).

constexpr month_day
operator/(const month& m, const day& d) noexcept;

Returns: {m, d}.

constexpr month_day
operator/(const month& m, int d) noexcept;

Returns: m / day(d).

constexpr month_day
operator/(int m, const day& d) noexcept;

Returns: month(m) / d.

constexpr month_day
operator/(const day& d, const month& m) noexcept;

Returns: m / d.
constexpr month_day
operator/(const day& d, int m) noexcept;
Returns: month(m) / d.

constexpr month_day_last
operator/(const month& m, last_spec) noexcept;
Returns: month_day_last(m).

constexpr month_day_last
operator/(int m, last_spec) noexcept;
Returns: month(m) / last.

constexpr month_day_last
operator/(last_spec, const month& m) noexcept;
Returns: m / last.

constexpr month_day_last
operator/(last_spec, int m) noexcept;
Returns: month(m) / last.

constexpr month_weekday
operator/(const month& m, const weekday_indexed& wdi) noexcept;
Returns: {m, wdi}.

constexpr month_weekday
operator/(int m, const weekday_indexed& wdi) noexcept;
Returns: month(m) / wdi.

constexpr month_weekday
operator/(const weekday_indexed& wdi, const month& m) noexcept;
Returns: m / wdi.

constexpr month_weekday
operator/(const weekday_indexed& wdi, int m) noexcept;
Returns: month(m) / wdi.

constexpr month_weekday_last
operator/(const month& m, const weekday_last& wdl) noexcept;
Returns: {m, wdl}.

constexpr month_weekday_last
operator/(int m, const weekday_last& wdl) noexcept;
Returns: month(m) / wdl.

constexpr month_weekday_last
operator/(const weekday_last& wdl, const month& m) noexcept;
Returns: m / wdl.

constexpr month_weekday_last
operator/(const weekday_last& wdl, int m) noexcept;
Returns: month(m) / wdl.

constexpr year_month_day
operator/(const year_month& ym, const day& d) noexcept;
Returns: {ym.year(), ym.month(), d}.

constexpr year_month_day
operator/(const year_month& ym, int d) noexcept;
Returns: ym / day(d).
constexpr year_month_day
    operator/(const year& 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()}}.

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 \texttt{time\_of\_day} \[time.tod\]

27.9.1 Overview \[time.tod.overview\]

namespace std::chrono {
    template<class Duration> class time_of_day;
    template<> class time_of_day<hours>;
    template<> class time_of_day<minutes>;
    template<> class time_of_day<seconds>;
    template<class Rep, class Period> class time_of_day<duration<Rep, Period>>;
}

The \texttt{time\_of\_day} class template splits a \texttt{duration} representing the time elapsed since midnight into a “broken down” time of day such as \texttt{hours:minutes:seconds}. The \texttt{Duration} template parameter dictates the precision to which the time is broken down. \[Note: This can vary from a coarse precision of hours to a very fine precision of nanoseconds. — end note\] A \texttt{time\_of\_day} object also tracks whether it should be output as a 12-hour time format or a 24-hour time format.

2 The primary \texttt{time\_of\_day} template is not defined. Four specializations are provided to handle four different levels of precision.

3 Each specialization of \texttt{time\_of\_day} is a trivially copyable and standard-layout class type.

27.9.2 Hours precision \[time.tod.hours\]

namespace std::chrono {
    template<> class time_of_day<hours> {
        public:
            using precision = chrono::hours;

            time_of_day() = default;
            constexpr explicit time_of_day(chrono::hours since_midnight) noexcept;

            constexpr chrono::hours hours() const noexcept;
            constexpr precision to_duration() const noexcept;

            constexpr void make24() noexcept;
            constexpr void make12() noexcept;
    };
}

\[Note: This specialization handles hours since midnight. — end note\]

constexpr explicit time_of_day(chrono::hours since_midnight) noexcept;

Effects: Constructs an object of type \texttt{time\_of\_day} in 24-hour format corresponding to \texttt{since\_midnight} hours after 00:00:00.
Ensures: hours() returns the integral number of hours since_midnight is after 00:00:00.

constexpr chrono::hours hours() const noexcept;

Returns: The stored hour of *this.

constexpr explicit operator precision() const noexcept;

Returns: The number of hours since midnight.

constexpr precision to_duration() const noexcept;

Returns: precision{*this}.

constexpr void make24() noexcept;

Effects: If *this is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.

constexpr void make12() noexcept;

Effects: If *this is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.

27.9.3 Minutes precision [time.tod.minutes]

namespace std::chrono {
    template<>
    class time_of_day<minutes> {
        public:
            using precision = chrono::minutes;
            time_of_day() = default;
            constexpr explicit time_of_day(chrono::minutes since_midnight) noexcept;
            constexpr chrono::hours hours() const noexcept;
            constexpr chrono::minutes minutes() const noexcept;
            constexpr explicit operator precision() const noexcept;
            constexpr precision to_duration() const noexcept;
            constexpr void make24() noexcept;
            constexpr void make12() noexcept;
    };
}

[Note: This specialization handles hours and minutes since midnight. — end note]

constexpr explicit time_of_day(minutes since_midnight) noexcept;

Effects: Constructs an object of type time_of_day in 24-hour format corresponding to since_midnight minutes after 00:00:00.

Ensures: hours() returns the integral number of hours since_midnight is after 00:00:00. minutes() returns the integral number of minutes since_midnight is after (00:00:00 + hours()).

constexpr chrono::hours hours() const noexcept;

Returns: The stored hour of *this.

constexpr chrono::minutes minutes() const noexcept;

Returns: The stored minute of *this.

constexpr explicit operator precision() const noexcept;

Returns: The number of minutes since midnight.

constexpr precision to_duration() const noexcept;

Returns: precision{*this}.

constexpr void make24() noexcept;

Effects: If *this is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.
constexpr void make12() noexcept;

Effects: If *this is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.

### 27.9.4 Seconds precision

```cpp
namespace std::chrono {
    template<class Rep, class Period>
    class time_of_day<duration<Rep, Period>> {
        public:
            using precision = chrono::seconds;
            time_of_day() = default;
            constexpr explicit time_of_day(chrono::seconds since_midnight) noexcept;
            constexpr chrono::hours hours() const noexcept;
            constexpr chrono::minutes minutes() const noexcept;
            constexpr chrono::seconds seconds() const noexcept;
            constexpr explicit operator precision() const noexcept;
            constexpr precision to_duration() const noexcept;
            constexpr void make24() noexcept;
            constexpr void make12() noexcept;
    };
}
```

1 [Note: This specialization handles hours, minutes, and seconds since midnight. — end note]

constexpr explicit time_of_day(seconds since_midnight) noexcept;

Effects: Constructs an object of type `time_of_day` in 24-hour format corresponding to `since_midnight` seconds after 00:00:00.

Ensures: `hours()` returns the integral number of hours `since_midnight` is after 00:00:00. `minutes()` returns the integral number of minutes `since_midnight` is after (00:00:00 + `hours()`). `seconds()` returns the integral number of seconds `since_midnight` is after (00:00:00 + `hours()` + `minutes()`).

```cpp
constexpr chrono::hours hours() const noexcept;
Returns: The stored hour of *this.
```

```cpp
constexpr chrono::minutes minutes() const noexcept;
Returns: The stored minute of *this.
```

```cpp
constexpr chrono::seconds seconds() const noexcept;
Returns: The stored second of *this.
```

```cpp
constexpr explicit operator precision() const noexcept;
Returns: The number of seconds since midnight.
```

```cpp
constexpr precision to_duration() const noexcept;
Returns: `precision{*this}`.
```

```cpp
constexpr void make24() noexcept;
Effects: If *this is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.
```

```cpp
constexpr void make12() noexcept;
Effects: If *this is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.
```

### 27.9.5 Sub-second precision

```cpp
namespace std::chrono {
    template<class Rep, class Period>
    class time_of_day<duration<Rep, Period>> {
        public:
```

§ 27.9.5
using precision = duration<Rep, Period>;

time_of_day() = default;
constexpr explicit time_of_day(precision since_midnight) 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;

constexpr void make24() noexcept;
constexpr void make12() noexcept;

1 This specialization shall not exist unless treat_as_floating_point_v<Rep> is false and duration<Rep, Period> is not convertible to seconds. [Note: This specialization handles hours, minutes, seconds, and fractional seconds since midnight. Typical uses are with milliseconds, microseconds and nanoseconds. —end note]

2 constexpr explicit time_of_day(precision since_midnight) noexcept;

Effects: Constructs an object of type time_of_day in 24-hour format corresponding to since_midnight fractional seconds after 00:00:00.

Ensures: hours() returns the integral number of hours since_sine_midnight is after 00:00:00. minutes() returns the integral number of minutes since_midnight is after (00:00:00 + hours()). seconds() returns the integral number of seconds since_midnight is after (00:00:00 + hours() + minutes()). subseconds() returns the integral number of fractional seconds since_midnight is after (00:00:00 + hours() + minutes() + seconds()).

3 constexpr chrono::hours hours() const noexcept;

Returns: The stored hour of *this.

4 constexpr chrono::minutes minutes() const noexcept;

Returns: The stored minute of *this.

5 constexpr chrono::seconds seconds() const noexcept;

Returns: The stored second of *this.

6 constexpr duration<Rep, Period> subseconds() const noexcept;

Returns: The stored subsecond of *this.

7 constexpr explicit operator precision() const noexcept;

Returns: The number of subseconds since midnight.

8 constexpr precision to_duration() const noexcept;

Returns: precision(*this).

9 constexpr void make24() noexcept;

Effects: If *this is a 12-hour time, converts to a 24-hour time. Otherwise, no effects.

10 constexpr void make12() noexcept;

Effects: If *this is a 24-hour time, converts to a 12-hour time. Otherwise, no effects.
operator<<(basic_ostream<charT, traits>& os, const time_of_day<hours>& t);

Effects: If t is a 24-hour time, outputs to os according to the format "%H00" (27.11). Otherwise outputs to os according to the format "%I%p" (27.11).

Returns: os.

[Example:
for (hours h : {1h, 18h}) {
    time_of_day<hours> tod(h);
    os << tod << 'n';
    tod.make12();
    os << tod << 'n';
}

Produces the output:
0100
1am
1800
6pm
— end example]

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const time_of_day<minutes>& t);

Effects: If t is a 24-hour time, outputs to os according to the format "%H:%M" (27.11). Otherwise outputs to os according to the format "%I:%M%p" (27.11).

Returns: os.

[Example:
for (minutes m : {68min, 1095min}) {
    time_of_day<minutes> tod(m);
    os << tod << 'n';
    tod.make12();
    os << tod << 'n';
}

Produces the output:
01:08
1:08am
18:15
6:15pm
— end example]

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const time_of_day<seconds>& t);

Effects: If t is a 24-hour time, outputs to os according to the format "%T" (27.11). Otherwise outputs to os according to the format "%I:%M:%S%p" (27.11).

Returns: os.

[Example:
for (seconds s : {4083s, 65745s}) {
    time_of_day<seconds> tod(s);
    os << tod << 'n';
    tod.make12();
    os << tod << 'n';
}

Produces the output:
01:08:03
1:08:03am

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const time_of_day<duration<Rep, Period>>& t);

Effects: If \( t \) is a 24-hour time, outputs to \( os \) according to the format "%T" (27.11). Otherwise outputs to \( os \) according to the format "%I:%M:%S%p" (27.11).

Returns: \( os \).

Example:

```cpp
for (milliseconds ms : {4083007ms, 65745123ms}) {
    time_of_day<seconds> tod(ms);
    os << tod << '
';
    tod.make12();
    os << tod << '
';
}
```

Produces the output:

01:08:03.007 1:08:03.007 am
18:15:45.123 6:15:45.123 pm

—end example—

### 27.10 Time zones

#### 27.10.1 In general

27.10 describes an interface for accessing the IANA Time Zone database described in RFC 6557, 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.

#### 27.10.2 Time zone database

Each `vector` in a `tzdb` object is sorted to enable fast lookup.

```cpp
const time_zone* locate_zone(string_view tz_name) const;
```

Returns: If a `time_zone` is found for which `name() == tz_name`, returns a pointer to that `time_zone`. Otherwise if a link is found for which `tz_name == link.name()`, then a pointer is returned to the `time_zone` for which `zone.name() == link.target()`. \[Note: A link specifies an alternative name for a time zone. —end note\]

Throws: If a `const time_zone*` cannot be found as described in the `Returns`: clause, throws a `runtime_error`. \[Note: On non-exceptional return, the return value is always a pointer to a valid `time_zone`. —end note\]

```cpp
const time_zone* current_zone() const;
```

Returns: A pointer to the time zone which the computer has set as its local time zone.
27.10.2.2 Class tzdb_list

namespace std::chrono {
class tzdb_list {
public:
    tzdb_list(const tzdb_list&) = delete;
    tzdb_list& operator=(const tzdb_list&) = delete;

    // unspecified additional constructors
    class const_iterator;
    const tzdb& front() const noexcept;
    const_iterator erase_after(const_iterator p);
    const_iterator begin() const noexcept;
    const_iterator end() const noexcept;
    const_iterator cbegin() const noexcept;
    const_iterator cend() const noexcept;
};
}

The tzdb_list database is a singleton; the unique object of type tzdb_list can be accessed via the get_tzdb_list() function. [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.

const tzdb& front() const noexcept;

Returns: A reference to the first tzdb in the container.

Remarks: This operation is thread-safe with respect to reload_tzdb(). [Note: reload_tzdb() pushes a new tzdb onto the front of this container. — end note]

const_iterator erase_after(const_iterator p);

Requires: 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.

Remarks: 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.

const_iterator begin() const noexcept;

Returns: An iterator referring to the first tzdb in the container.

const_iterator end() const noexcept;

Returns: An iterator referring to the position one past the last tzdb in the container.

const_iterator cbegin() const noexcept;

Returns: begin().

const_iterator cend() const noexcept;

Returns: end().
27.10.2.3 Time zone database access

`tzdb_list& get_tzdb_list();`

1. **Effects:** If this is the first access to the time zone database, initializes the database. If this call initializes the database, the resulting database will be a `tzdb_list` holding a single initialized `tzdb`.
2. **Returns:** A reference to the database.
3. **Remarks:** It is safe to call this function from multiple threads at one time.
4. **Throws:** `runtime_error` if for any reason a reference cannot be returned to a valid `tzdb_list` containing one or more valid `tzdb`s.

`const tzdb& get_tzdb();`

5. **Returns:** `get_tzdb_list().front()`.

`const time_zone* locate_zone(string_view tz_name);`

6. **Returns:** `get_tzdb().locate_zone(tz_name)`.

[Note: The time zone database will be initialized if this is the first reference to the database. — end note]

`const time_zone* current_zone();`

7. **Returns:** `get_tzdb().current_zone()`.

27.10.2.4 Remote time zone database support

The local time zone database is that supplied by the implementation when the program first accesses the database, for example via `current_zone()`. While the program is running, the implementation may choose to update the time zone database. This update shall not impact the program in any way unless the program calls the functions in this subclause. This potentially updated time zone database is referred to as the remote time zone database.

`const tzdb& reload_tzdb();`

2. **Effects:** This function first checks the version of the remote time zone database. If the versions of the local and remote databases are the same, there are no effects. Otherwise the remote database is pushed to the front of the `tzdb_list` accessed by `get_tzdb_list()`.
3. **Returns:** `get_tzdb_list().front()`.
4. **Remarks:** No pointers, references, or iterators are invalidated.
5. **Remarks:** This function is thread-safe with respect to `get_tzdb_list().front()` and `get_tzdb_list().erase_after()`.
6. **Throws:** `runtime_error` if for any reason a reference cannot be returned to a valid `tzdb`.

`string remote_version();`

7. **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]

27.10.3 Exception classes

27.10.3.1 Class nonexistent_local_time

```cpp
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);
    };
}
```

1. **nonexistent_local_time** is thrown when an attempt is made to convert a non-existent `local_time` to a `sys_time` without specifying `choose::earliest` or `choose::latest`. 

§ 27.10.3.1
template<class Duration>
nonexistent_local_time(const local_time<Duration>& tp, const local_info& i);

Requires: i.result == local_info::nonexistent.

Effects: Constructs a nonexistent_local_time by initializing the base class with a sequence of char equivalent to that produced by os.str() initialized as shown below:

istringstream os;
os << tp << " is in a gap between\n" << local_seconds(i.first.end.time_since_epoch()) + i.first.offset << ', '
<< i.first.abbrev << " and\n" << local_seconds(i.second.begin.time_since_epoch()) + i.second.offset << ',
<< i.second.abbrev
<< " which are both equivalent to\n" << i.first.end << " UTC";

Example:
#include <chrono>
#include <iostream>

int main() {
  using namespace std::chrono;
  try {
    auto zt = zoned_time("America/New_York",
      local_days{Sunday[2]/March/2016} + 2h + 30min);
  } catch (const nonexistent_local_time& e) {
    std::cout << e.what() << '\n';
  }
}

Produces the output:
2016-03-13 02:30:00 is in a gap between
2016-03-13 02:00:00 EST and
2016-03-13 03:00:00 EDT which are both equivalent to
2016-03-13 07:00:00 UTC

— end example]
#include <chrono>
#include <iostream>

int main() {
    using namespace std::chrono;
    try {
        auto zt = zoned_time("America/New_York",
            local_days{Sunday[1]/November/2016} + 1h + 30min);
    } catch (const ambiguous_local_time& e) {
        std::cout << e.what() << 'n';
    }
}

Produces the output:

2016-11-06 01:30:00 is ambiguous. It could be
2016-11-06 01:30:00 EDT == 2016-11-06 05:30:00 UTC or
2016-11-06 01:30:00 EST == 2016-11-06 06:30:00 UTC

— end example]

27.10.4 Information classes [time.zone.info]
27.10.4.1 Class sys_info [time.zone.info.sys]

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:

    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.
27.10.4.2 Class `local_info` [time.zone.info.local]

```cpp
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;
    };
}
```

1 [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]

2 Describes the result of converting a `local_time` to a `sys_time` as follows:

   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. 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`.
   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`.

```cpp
    template<class charT, class traits>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const local_info& r);
```  

3 Streams out the `local_info` object `r` in an unspecified format.

27.10.5 Class `time_zone` [time.zone.timezone]

27.10.5.1 Overview [time.zone.overview]

```cpp
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;  
    };  
}
```

§ 27.10.5.1 1255
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]

### 27.10.5.2 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`.

### 27.10.5.3 Non-member functions

**bool operator==(const time_zone& x, const time_zone& y) noexcept;**

- **Returns:** `x.name() == y.name()`.

**bool operator<(const time_zone& x, const time_zone& y) noexcept;**

- **Returns:** `x.name() < y.name()`.

### 27.10.6 Class template zoned_traits

**namespace std::chrono {**

```cpp
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:

```cpp
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: std::chrono::locate_zone("UTC").

static const time_zone* locate_zone(string_view name);

Returns: std::chrono::locate_zone(name).

27.10.7 Class template zoned_time

27.10.7.1 Overview

namespace std::chrono {

template<class Duration, class TimeZonePtr = const time_zone*>(

class zoned_time {

public:

using duration = common_type_t<Duration, seconds>;

private:

TimeZonePtr zone_; // exposition only
sys_time<duration> tp_; // exposition only

using traits = zoned_traits<TimeZonePtr>;
// exposition only

public:

zoned_time();

zoned_time(const zoned_time&) = default;

zoned_time& operator=(const zoned_time&) = default;

zoned_time(const sys_time<Duration>& st);

explicit zoned_time(TimeZonePtr z);

explicit zoned_time(string_view name);

template<class Duration2>

zoned_time(const zoned_time<Duration2, TimeZonePtr>& zt) noexcept;

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);

zoned_time(string_view name, const zoned_time<Duration>& zt);

zoned_time(string_view name, const zoned_time<Duration>& zt, choose);

zoned_time& operator=(const sys_time<Duration>& st);

zoned_time& operator=(const local_time<Duration>& ut);

operator sys_time<duration>() const;

explicit operator local_time<duration>() const;

TimeZonePtr get_time_zone() const;

local_time<duration> get_local_time() const;

sys_time<duration> get_sys_time() const;

sys_info get_info() const;

};
zoned_time() \to zoned_time<seconds>;

\text{template}<\text{class } \text{Duration}>\n\text{zoned_time(sys_time<Duration>)}
\to zoned_time<\text{common_type_t<Duration, seconds>>};

\text{template}<\text{class TimeZonePtr, class Duration}>
\text{zoned_time(TimeZonePtr, sys_time<Duration>)}
\to zoned_time<\text{common_type_t<Duration, seconds>>, TimeZonePtr>;

\text{template}<\text{class TimeZonePtr, class Duration}>
\text{zoned_time(TimeZonePtr, local_time<Duration>, choose = choose::earliest)}
\to zoned_time<\text{common_type_t<Duration, seconds>>, TimeZonePtr>;

\text{template}<\text{class TimeZonePtr, class Duration}>
\text{zoned_time(TimeZonePtr, zoned_time<Duration>, choose = choose::earliest)}
\to zoned_time<\text{common_type_t<Duration, seconds>>, TimeZonePtr>;

zoned_time(string_view) \to zoned_time<seconds>;

\text{template}<\text{class Duration}>
\text{zoned_time(string_view, sys_time<Duration>)}
\to zoned_time<\text{common_type_t<Duration, seconds>>};

\text{template}<\text{class Duration}>
\text{zoned_time(string_view, local_time<Duration>, choose = choose::earliest)}
\to zoned_time<\text{common_type_t<Duration, seconds>>};

\text{template}<\text{class Duration, class TimeZonePtr, class TimeZonePtr2}>
\text{zoned_time(TimeZonePtr, zoned_time(Duration, TimeZonePtr2>, choose = choose::earliest)}
\to zoned_time<Duration, TimeZonePtr>;

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.

### 27.10.7.2 Constructors

[zoned_time.ctor]

\text{zoned_time();}

1 \textit{Remarks:} This constructor does not participate in overload resolution unless traits::default_zone() is a well-formed expression.

2 \textit{Effects:} Constructs a zoned_time by initializing \text{zone_} with traits::default_zone() and default constructing \text{tp_}.

\text{zoned_time(const sys_time<Duration>& st);}  

3 \textit{Remarks:} This constructor does not participate in overload resolution unless traits::default_zone() is a well-formed expression.

4 \textit{Effects:} Constructs a zoned_time by initializing \text{zone_} with traits::default_zone() and \text{tp_} with \text{st}.

\text{explicit zoned_time(TimeZonePtr z);}  

5 \textit{Requires:} \text{z} refers to a time zone.

6 \textit{Effects:} Constructs a zoned_time by initializing \text{zone_} with std::move(z).

\text{explicit zoned_time(string_view name);}  

7 \textit{Remarks:} This constructor does not participate in overload resolution unless traits::locate_zone(string_view{}) is a well-formed expression and zoned_time is constructible from the return type of traits::locate_zone(string_view{}).
Effects: Constructs a zoned_time by initializing zone_ with traits::locate_zone(name) and default constructing tp_.

template<class Duration2>
zoned_time(const zoned_time<Duration2, TimeZonePtr>& y) noexcept;

Remarks: Does not participate in overload resolution unless sys_time<Duration> is implicitly convertible to sys_time<Duration>.

Effects: Constructs a zoned_time by initializing zone_ with y.zone_ and tp_ with y.tp_.

zoned_time(TimeZonePtr z, const sys_time<Duration>& st);

Requires: z refers to a time zone.

Effects: Constructs a zoned_time by initializing zone_ with std::move(z) and tp_ with st.

zoned_time(string_view name, const sys_time<Duration>& st);

Remarks: This constructor does not participate in overload resolution unless zoned_time is constructible from the return type of traits::locate_zone(name) and st.

Effects: Equivalent to construction with {traits::locate_zone(name), st}.

zoned_time(TimeZonePtr z, const local_time<Duration>& tp);

Requires: z refers to a time zone.

Remarks: This constructor does not participate in overload resolution unless decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>{})) is convertible to sys_time<duration>.

Effects: Constructs a zoned_time by initializing zone_ with std::move(z) and tp_ with zone_->to_sys(tp).

zoned_time(string_view name, const local_time<Duration>& tp);

Remarks: This constructor does not participate in overload resolution unless zoned_time is constructible from the return type of traits::locate_zone(name) and tp.

Effects: Equivalent to construction with {traits::locate_zone(name), st}.

zoned_time(TimeZonePtr z, const local_time<Duration>& tp, choose c);

Requires: z refers to a time zone.

Remarks: This constructor does not participate in overload resolution unless decltype(declval<TimeZonePtr&>()->to_sys(local_time<Duration>(), choose::earliest)) is convertible to sys_time<duration>.

Effects: Constructs a zoned_time by initializing zone_ with std::move(z) and tp_ with zone_->to_sys(tp, c).

zoned_time(string_view name, const local_time<Duration>& tp, choose c);

Remarks: This constructor does not participate in overload resolution unless zoned_time is constructible from the return type of traits::locate_zone(name), local_time<Duration>, and choose.

Effects: Equivalent to construction with {traits::locate_zone(name), tp, c}.

template<class Duration2, class TimeZonePtr2>
zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y);

Remarks: Does not participate in overload resolution unless sys_time<Duration> is implicitly convertible to sys_time<Duration>.

Requires: z refers to a valid time zone.

Effects: Constructs a zoned_time by initializing zone_ with std::move(z) and tp_ with y.tp_.
27.10.7.3 Member functions

\[
zoned_time(TimeZonePtr z, const zoned_time<Duration2, TimeZonePtr2>& y, choose);
\]
Remarks: Does not participate in overload resolution unless \texttt{sys_time<Duration2> is implicitly convertible to sys_time<Duration>}.
Requires: \( z \) refers to a valid time zone.
Effects: Equivalent to construction with \( \{z, y\} \).
[Note: The \texttt{choose} parameter has no effect. — end note]

\[
zoned_time(string_view name, const zoned_time<Duration>& y);
\]
Remarks: This constructor does not participate in overload resolution unless \texttt{zoned_time} is constructible from the return type of \texttt{traits::locate_zone(name) and zoned_time}.
Effects: Equivalent to construction with \( \{\texttt{traits::locate_zone(name)}, y\} \).

\[
zoned_time(string_view name, const zoned_time<Duration>& y, choose c);
\]
Remarks: This constructor does not participate in overload resolution unless \texttt{zoned_time} is constructible from the return type of \texttt{traits::locate_zone(name), zoned_time, and choose}.
Effects: Equivalent to construction with \( \{\texttt{traits::locate_zone(name)}, y, c\} \).
[Note: The \texttt{choose} parameter has no effect. — end note]

27.10.7.4 Non-member functions

\[
\text{template<class Duration1, class Duration2, class TimeZonePtr>}
\text{bool operator==(const zoned_time<Duration1, TimeZonePtr>& x,}
\text{ const zoned_time<Duration2, TimeZonePtr>& y)};
\]
Returns: \( x.zone_ == y.zone_ && x.tp_ == y.tp_ \).
template<class Duration1, class Duration2, class TimeZonePtr>
bool operator!=(const zoned_time<Duration1, TimeZonePtr>& x,
const zoned_time<Duration2, TimeZonePtr>& y);

Returns: !(x == y).

Effects: Streams the value returned from t.get_local_time() to os using the format "%F %T %Z".

Returns: os.

Effects: First obtains a sys_info via tp.get_info() which for exposition purposes will be referred to as info. Then calls to_stream(os, fmt, tp.get_local_time(), &info.abbrev, &info.offset).

Returns: os.

27.10.8 Class leap

27.10.8.1 Overview

namespace std::chrono {

    class leap {
    public:
        leap(const leap&) = default;
        leap& operator=(const leap&) = default;

        // unspecified additional constructors

        constexpr sys_seconds date() const noexcept;
    }
}

Objects of type leap representing the date of the leap second insertions are constructed and stored in the time zone database when initialized.

Example:

for (auto l : get_tzdb().leaps)
    if (l <= 2018y/March/17d)
        cout << l.date() << '\n';

Produces the output:

1972-07-01 00:00:00
1973-01-01 00:00:00
1974-01-01 00:00:00
1975-01-01 00:00:00
1976-01-01 00:00:00
1977-01-01 00:00:00
1978-01-01 00:00:00
1979-01-01 00:00:00
1980-01-01 00:00:00
1981-07-01 00:00:00
1982-07-01 00:00:00
1983-07-01 00:00:00
1985-07-01 00:00:00
1988-01-01 00:00:00
1990-01-01 00:00:00
1991-01-01 00:00:00
1992-07-01 00:00:00
1993-07-01 00:00:00
1994-07-01 00:00:00

§ 27.10.8.1
27.10.8.2 Member functions

```cpp
constexpr sys_seconds date() const noexcept;
```

Returns: The date and time at which the leap second was inserted.

27.10.8.3 Non-member functions

```cpp
constexpr bool operator==(const leap& x, const leap& y) noexcept;
```

Returns: `x.date() == y.date()`.

```cpp
constexpr bool operator<(const leap& x, const leap& y) noexcept;
```

Returns: `x.date() < y.date()`.

```cpp
template<class Duration>
constexpr bool operator==(const leap& x, const sys_time<Duration>& y) noexcept;
```

Returns: `x.date() == y`.

```cpp
template<class Duration>
constexpr bool operator==(const sys_time<Duration>& x, const leap& y) noexcept;
```

Returns: `y == x`.

```cpp
template<class Duration>
constexpr bool operator!=(const leap& x, const sys_time<Duration>& y) noexcept;
```

Returns: `!(x == y)`.

```cpp
template<class Duration>
constexpr bool operator!=(const sys_time<Duration>& x, const leap& y) noexcept;
```

Returns: `!(x == y)`.

```cpp
template<class Duration>
constexpr bool operator<(const leap& x, const sys_time<Duration>& y) noexcept;
```

Returns: `x.date() < y`.

```cpp
template<class Duration>
constexpr bool operator<(const sys_time<Duration>& x, const leap& y) noexcept;
```

Returns: `x < y.date()`.

```cpp
template<class Duration>
constexpr bool operator>(const leap& x, const sys_time<Duration>& y) noexcept;
```

Returns: `y < x`.

```cpp
template<class Duration>
constexpr bool operator>(const sys_time<Duration>& x, const leap& y) noexcept;
```

Returns: `y < x`.

```cpp
template<class Duration>
constexpr bool operator<=(const leap& x, const sys_time<Duration>& y) noexcept;
```

Returns: `!(y < x)`.

§ 27.10.8.3
template<class Duration>
constexpr bool operator<=(const sys_time<Duration>& x, const leap& y) noexcept;

Returns: !(y < x).

template<class Duration>
constexpr bool operator>=(const leap& x, const sys_time<Duration>& y) noexcept;

Returns: !(x < y).

template<class Duration>
constexpr bool operator>=(const sys_time<Duration>& x, const leap& y) noexcept;

Returns: !(x < y).

27.10.9 Class link

27.10.9.1 Overview

namespace std::chrono {
    class link {
        public:
            link(link&&) = default;
            link& operator=(link&&) = default;

            // unspecified additional constructors

            string_view name() const noexcept;
            string_view target() const noexcept;
    };
}

A link specifies an alternative name for a time_zone. links are constructed when the time zone database is initialized.

27.10.9.2 Member functions

string_view name() const noexcept;

Returns: The alternative name for the time zone.

string_view target() const noexcept;

Returns: The name of the time_zone for which this link provides an alternative name.

27.10.9.3 Non-member functions

bool operator==(const link& x, const link& y) noexcept;

Returns: x.name() == y.name().

bool operator<(const link& x, const link& y) noexcept;

Returns: x.name() < y.name().

27.11 Formatting

Each format overload specified in this subclause calls to_stream unqualified, so as to enable argument dependent lookup (6.4.2).

template<class charT, class Streamable>
basic_string<charT> format(const charT* fmt, const Streamable& s);

Remarks: This function shall not participate in overload resolution unless
to_stream(declval<basic_ostream<charT>&>(), fmt, s)
is a valid expression.

Effects: Constructs a local variable of type basic_ostringstream<charT> (named os for exposition purposes). Executes os.exceptions(ios::failbit | ios::badbit). Then calls to_stream(os, fmt, s).
Returns: \texttt{os.str()}.

\begin{verbatim}
template<class charT, class Streamable>
  basic_string<charT>
  format(const locale& loc, const charT* fmt, const Streamable& s);
\end{verbatim}

Remarks: This function shall not participate in overload resolution unless
to_stream(declval<basic_ostream<charT>&>(), fmt, s)
is a valid expression.

Effects: Constructs a local variable of type basic_ostringstream<charT> (named \texttt{os} for exposition purposes). Executes \texttt{os.exceptions(ios::failbit | ios::badbit)}. Then calls \texttt{os.imbue(loc)}. Then calls to_stream(os, fmt, s).

Returns: \texttt{os.str()}.

\begin{verbatim}
template<class charT, class traits, class Alloc, class Streamable>
  basic_string<charT, traits, Alloc>
  format(const basic_string<charT, traits, Alloc>& fmt, const Streamable& s);
\end{verbatim}

Remarks: This function shall not participate in overload resolution unless
to_stream(declval<basic_ostringstream<charT, traits, Alloc>&>(), fmt.c_str(), s)
is a valid expression.

Effects: Constructs a local variable of type basic_ostringstream<charT, traits, Alloc> (named \texttt{os} for exposition purposes). Executes \texttt{os.exceptions(ios::failbit | ios::badbit)}. Then calls to_stream(os, fmt.c_str(), s).

Returns: \texttt{os.str()}.

\begin{verbatim}
template<class charT, class traits, class Alloc, class Streamable>
  basic_string<charT, traits, Alloc>
  format(const locale& loc, const basic_string<charT, traits, Alloc>& fmt, const Streamable& s);
\end{verbatim}

Remarks: This function shall not participate in overload resolution unless
to_stream(declval<basic_ostringstream<charT, traits, Alloc>&>(), fmt.c_str(), s)
is a valid expression.

Effects: Constructs a local variable of type basic_ostringstream<charT, traits, Alloc> (named \texttt{os} for exposition purposes). Then calls \texttt{os.imbue(loc)}. Executes \texttt{os.exceptions(ios::failbit | ios::badbit)}. Then calls to_stream(os, fmt.c_str(), s).

Returns: \texttt{os.str()}.

The \textit{format} functions call a \textit{to_stream} function with a basic_ostream, a formatting string specifier, and a Streamable argument. Each \textit{to_stream} overload is customized for each Streamable type. However all \textit{to_stream} overloads treat the formatting string specifier according to the following specification:

The \textit{fmt} string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a \% character, possibly followed by an \texttt{E} or \texttt{O} modifier character (described below), followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (excluding the terminating null character) are streamed unchanged into the basic_ostream.

Each conversion specifier is replaced by appropriate characters as described in Table 87. Some of the conversion specifiers depend on the locale which is imbued to the basic_ostream. If the Streamable object does not contain the information the conversion specifier refers to, the value streamed to the basic_ostream is unspecified.

Unless explicitly specified, Streamable types will not contain time zone abbreviation and time zone offset information. If available, the conversion specifiers \%Z and \%z will format this information (respectively). If the information is not available, and \%Z or \%z are contained in \textit{fmt}, \texttt{os.setstate(ios_base::failbit)} shall be called.
Table 87 — Meaning of format conversion specifiers

<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, <code>setstate(ios::failbit)</code> is called.</td>
</tr>
<tr>
<td>%A</td>
<td>The locale’s full weekday name. If the value does not contain a valid weekday, <code>setstate(ios::failbit)</code> is called.</td>
</tr>
<tr>
<td>%b</td>
<td>The locale’s abbreviated month name. If the value does not contain a valid month, <code>setstate(ios::failbit)</code> is called.</td>
</tr>
<tr>
<td>%B</td>
<td>The locale’s full month name. If the value does not contain a valid month, <code>setstate(ios::failbit)</code> is called.</td>
</tr>
<tr>
<td>%c</td>
<td>The locale’s date and time representation. The modified command <code>%Ec</code> produces the locale’s alternate date and time representation.</td>
</tr>
<tr>
<td>%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 <code>%EC</code> 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 <code>%Od</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%D</td>
<td>Equivalent to <code>%m/%d/%y</code>.</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 <code>%Oe</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%F</td>
<td>Equivalent to <code>%Y-%m-%d</code>.</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 <code>%OH</code> 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 <code>%OI</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%j</td>
<td>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 <code>%Om</code> 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 <code>%OM</code> produces the locale’s alternative representation.</td>
</tr>
<tr>
<td>%n</td>
<td>A new-line character.</td>
</tr>
<tr>
<td>%p</td>
<td>The locale’s equivalent of the AM/PM designations associated with a 12-hour clock.</td>
</tr>
<tr>
<td>%r</td>
<td>The locale’s 12-hour clock time.</td>
</tr>
<tr>
<td>%t</td>
<td>A horizontal-tab character.</td>
</tr>
<tr>
<td>%T</td>
<td>Equivalent to <code>%H:%M:%S</code>.</td>
</tr>
<tr>
<td>%u</td>
<td>The ISO weekday as a decimal number (1-7), where Monday is 1. The modified command <code>%0u</code> produces the locale’s alternative representation.</td>
</tr>
</tbody>
</table>
Table 87 — Meaning of format conversion specifiers (continued)

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<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 %OU 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 %OV 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 %Ow 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 %OW 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.</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.</td>
</tr>
<tr>
<td>%z</td>
<td>The offset from UTC in the ISO 8601 format. For example −0430 refers to 4 hours 30 minutes behind UTC. If the offset is zero, +0000 is used. The modified commands %Ez and %Oz insert a : between the hours and minutes: −04:30. If the offset information is not available, setstate(ios_base::failbit) shall be called.</td>
</tr>
<tr>
<td>%Z</td>
<td>The time zone abbreviation. If the time zone abbreviation is not available, setstate(ios_base::failbit) shall be called.</td>
</tr>
<tr>
<td>%%</td>
<td>A % character.</td>
</tr>
</tbody>
</table>

27.12 Parsing [time.parse]

1 Each parse overload specified in this subclause calls from_stream unqualified, so as to enable argument dependent lookup (6.4.2).

```cpp
template<class charT, class traits, class Alloc, class Parsable>
unspecified
parse(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp);
```

Remarks: This function shall not participate in overload resolution unless
```
from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp)
```
is a valid expression.

Returns: A manipulator that, when extracted from a basic_istream<charT, traits> is, calls from_stream(is, fmt.c_str(), tp).

```cpp
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);
```

Remarks: This function shall not participate in overload resolution unless
```
from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp, addressof(abbrev))
```
is a valid expression.

Returns: A manipulator that, when extracted from a basic_istream<charT, traits> is, calls from_stream(is, fmt.c_str(), tp, addressof(abbrev)).
template<class charT, class traits, class Alloc, class Parsable>
unspecified
parses(const basic_string<charT, traits, Alloc>& fmt, Parsable& tp,
minutes & offset);

Remarks: This function shall not participate in overload resolution unless
from_stream(declval<basic_istream<charT, traits>&>(), fmt.c_str(), tp, nullptr, &offset)
is a valid expression.

Returns: A manipulator that, when extracted from a basic_istream<charT, traits> is, calls
from_stream(is, fmt.c_str(), tp, 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);

Remarks: This function shall not participate in overload resolution unless
from_stream(declval<basic_istream<charT, traits>&>(),
fmt.c_str(), tp, addressof(abbrev), &offset)
is a valid expression.

Returns: A manipulator that, when extracted from a basic_istream<charT, traits> is, calls
from_stream(is, fmt.c_str(), tp, addressof(abbrev), &offset).

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 the table below. 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 the table below, 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.

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 88 — 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 commands %EC and %OC interpret 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 %Ed 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>Flag</td>
<td>Parsed value</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>%G</td>
<td>The ISO week-based year as a decimal number. The modified command %G 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>
<tr>
<td>%H</td>
<td>The hour (24-hour clock) as a decimal number. The modified command %NH 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 %OH interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%I</td>
<td>The hour (12-hour clock) as a decimal number. The modified command %NI 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>%j</td>
<td>The day of the year as a decimal number. Jan 1 is 1. The modified command %Nj 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>%m</td>
<td>The month as a decimal number. Jan is 1. The modified command %Nm 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 %Om interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%M</td>
<td>The minutes as a decimal number. The modified command %NM 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 %OM interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%n</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 %t&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>%p</td>
<td>The locale’s equivalent of the AM/PM designations associated with a 12-hour clock. The command %I must precede %p in the format string.</td>
</tr>
<tr>
<td>%r</td>
<td>The locale’s 12-hour clock time.</td>
</tr>
<tr>
<td>%s</td>
<td>Equivalent to %H: %M.</td>
</tr>
<tr>
<td>%S</td>
<td>The seconds as a decimal number. The modified command %NS 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 %OS interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%t</td>
<td>Matches zero or one white space characters.</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 %U specifies the maximum number of characters to read. If ( N ) is not specified, the default is 1. Leading zeroes are permitted but not required. The modified command %OU interprets the locale’s alternative representation.</td>
</tr>
<tr>
<td>%v</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 %V 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>%w</td>
<td>The ISO week-based week number as a decimal number. The modified command %WV 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>%W</td>
<td>The ISO week-based week number as a decimal number. The modified command %WU 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>%d</td>
<td>The weekday as a decimal number (0-6), where Sunday is 0. The modified command %dw specifies the maximum number of characters to read. If ( N ) is not specified, the default is 1. Leading zeroes are permitted but not required. The modified command %Od interprets the locale’s alternative representation.</td>
</tr>
</tbody>
</table>
### Table 88 — 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.</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 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>

#### 27.13 Header <ctime> synopsis

```c
#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);
}
```

1 The contents of the header `<ctime>` are the same as the C standard library header `<time.h>`.

259) `strftime` supports the C conversion specifiers C, D, E, F, 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 (16.5.5.9).

See also: ISO C 7.27

\[^2\]
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 89.

Table 89 — Localization library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.3</td>
<td>&lt;locale&gt;</td>
</tr>
<tr>
<td>28.4</td>
<td>Standard locale Categories</td>
</tr>
<tr>
<td>28.5</td>
<td>&lt;locale&gt;</td>
</tr>
</tbody>
</table>

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> charT toupper(charT c, const locale& loc);
    template<class charT> charT tolower(charT c, const locale& loc);
}
```
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;
}

1 The header `<locale>` defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale. 260

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);

    260) In this subclause, the type name struct tm is an incomplete type that is defined in `<ctime>`.

§ 28.3.1
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
basic_string<char> name() const;

bool operator==(const locale& other) 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<>.

[Example: An iostream operator<< might be implemented as:]  

```cpp
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) {
        ios_base::iostate err = ios_base::iostate::goodbit;
        tm tmbuf; d.extract(tmbuf);
        use_facet<time_put<charT, ostreambuf_iterator<charT, traits>>>(s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
        s.setstate(err);  // 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 ctype 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.

261) Note that in the call to put the stream is implicitly converted to an ostreambuf_iterator<charT, traits>.
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 (16.5.5.9).

28.3.1.1 Types

28.3.1.1.1 Type `locale::category`

```
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 90.

<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 90, `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 91.

The provided implementation of members of facets `num_get<char>` and `num_put<char>` calls `use_facet<F>(1)` only for facet `F` of types `numpunct<char>` and `ctype<char>`, 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 satisfy the *C++17* `InputIterator` requirements or
Table 91 — Required specializations

<table>
<thead>
<tr>
<th>Category</th>
<th>Includes facets</th>
</tr>
</thead>
<tbody>
<tr>
<td>collate</td>
<td>collate_byname&lt;char&gt;, collate_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td>ctype</td>
<td>ctype_byname&lt;char&gt;, ctype_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char16_t, char8_t, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char32_t, char8_t, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;wchar_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>money_punct_byname&lt;char, International&gt;</td>
</tr>
<tr>
<td></td>
<td>money_punct_byname&lt;wchar_t, International&gt;</td>
</tr>
<tr>
<td></td>
<td>money_get&lt;C, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>money_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>numpunct_byname&lt;char&gt;, numpunct_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;C, InputIterator&gt;, num_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages_byname&lt;char&gt;, messages_byname&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>

*Cpp17OutputIterator* requirements, respectively (23.3). A template parameter with name `C` represents the set of types containing `char`, `wchar_t`, and any other implementation-defined character types that satisfy 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.

---

262) 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.

### 28.3.1.3 Class locale::id

```cpp
namespace std {
    class locale::id {
        public:
            id();
            void operator=(const id&) = delete;
            id(const id&) = delete;
        }
    }
}
```

1. The class `locale::id` provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

2. [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 (6.8.3.2). — end note]

### 28.3.1.2 Constructors and destructor

```cpp
locale() noexcept;
```

1. Default constructor: a snapshot of the current global locale.

2. 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()`.

3. [Note: This constructor is commonly used as the default value for arguments of functions that take a `const locale&` argument. — end note]

```cpp
locale(const locale& other) noexcept;
```

4. Effects: Constructs a locale which is a copy of `other`.

```cpp
explicit locale(const char* std_name);
```

5. Effects: Constructs a locale using standard C locale names, e.g., "POSIX". The resulting locale implements semantics defined to be associated with that name.

6. Remarks: The set of valid string argument values is "C", ",", and any implementation-defined values.

```cpp
explicit locale(const string& std_name);
```

7. Effects: The same as `locale(std_name.c_str())`.

```cpp
locale(const locale& other, const char* std_name, category);
```

8. 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)`.

9. 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);
```

10. Effects: The same as `locale(other, std_name.c_str(), cat)`.
template<class Facet> locale(const locale& other, Facet* f);

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.

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.

A non-virtual destructor that throws no exceptions.

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.

basic_string<char> 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.

bool operator!=(const locale& other) const;

Returns: !(*this == other).

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) satisfies 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 (25.7.1, 22.3.11):
std::sort(v.begin(), v.end(), loc);
— end example]
28.3.1.5 Static members

static locale global(const locale& loc);
1
Sets the global locale to its argument.

Effects: 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. No library function other
than locale::global() shall affect the value returned by locale(). [Note: See 28.5 for data race
considerations when setlocale is invoked. — end note]
3
Returns: The previous value of locale().

static const locale& classic();
4
The "C" locale.
5
Returns: A locale that implements the classic "C" locale semantics, equivalent to the value locale("C").
6
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);
1
Requires: Facet is a facet class whose definition contains the public static member id as defined
in 28.3.1.1.2.
2
Returns: A reference to the corresponding facet of loc, if present.
3
Throws: bad_cast if has_facet<Facet>(loc) is false.
4
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;
5
Returns: true if the facet requested is present in loc; otherwise false.

28.3.3 Convenience interfaces

28.3.3.1 Character classification

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 isgraph (charT c, const locale& loc);
template<class charT> bool isblank (charT c, const locale& loc);
1
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).\footnote{263}

28.3.3.2 Conversions

28.3.3.2.1 Character conversions

template<class charT> charT toupper(charT c, const locale& loc);
1
Returns: use_facet<ctype<charT>>(loc).toupper(c).

\footnote{263}{When used in a loop, it is faster to cache the \texttt{ctype<>} facet and use it directly, or use the vector form of \texttt{ctype<>::is}.}
template<class charT> char tolower(charT c, const locale& loc);

Returns: use_facet<ctype<charT>>(loc).tolower(c).

28.4 Standard locale categories

Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users’ iostream operators << and>>, as members put() and get(), respectively. 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.

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 {
    public:
        using mask = see below;

        // numeric values are for exposition only.
        static const mask space = 1 << 0;
        static const mask print = 1 << 1;
        static const mask cntrl = 1 << 2;
        static const mask upper = 1 << 3;
        static const mask lower = 1 << 4;
        static const mask alpha = 1 << 5;
        static const mask digit = 1 << 6;
        static const mask punct = 1 << 7;
        static const mask xdigit = 1 << 8;
        static const mask blank = 1 << 9;
        static const mask alnum = alpha | digit;
        static const mask graph = alnum | punct;
    }
}

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 {
    public:
        using char_type = charT;

        explicit ctype(size_t refs = 0);

        bool is(mask m, charT c) const;
        const charT* is(const charT* low, const charT* high, mask* vec) const;
        const charT* scan_is(mask m, const charT* low, const charT* high) const;
        const charT* scan_not(mask m, const charT* low, const charT* high) const;
        charT toupper(charT c) const;
        const charT* toupper(charT* low, const charT* high) const;
        charT tolower(charT c) const;
        const charT* tolower(charT* low, const charT* high) const;
        charT widen(char c) const;
        const char* widen(const char* low, const char* high, charT* to) const;
        char narrow(charT c, char dfault) const;
        const charT* narrow(const charT* low, const charT* high, char dfault, char* to) const;
    }
static locale::id id;

protected:

- ctype();
virtual bool do_is(mask m, charT c) const;
virtual const charT* do_is(const charT* low, const charT* high, mask* vec) const;
virtual const charT* do_scan_is(mask m, const charT* low, const charT* high) const;
virtual const charT* do_scan_not(mask m, const charT* low, const charT* high) const;
virtual charT do_toupper(charT) const;
virtual const charT* do_toupper(charT* low, const charT* high) const;
virtual charT do_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 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 90 (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) const;
const charT* toupper(charT* low, const charT* high) const;

4 Returns: do_toupper(c) or do_toupper(low, high).

charT 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).

char narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT* high, char dfault, char* to) const;

7 Returns: do_narrow(c, dfault) or do_narrow(low, high, dfault, to).

28.4.1.1.2 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].
Returns: The first form returns the result of the expression \((M & \text{m}) \neq 0\); i.e., \text{true} if the character has the characteristics specified. The second form returns \text{high}.

\begin{verbatim}
const char* do_scan_is(mask m, const charT* low, const charT* high) const;
\end{verbatim}

Effects: Locates a character in a buffer that conforms to a classification \text{m}.

Returns: The smallest pointer \(p\) in the range \([\text{low}, \text{high})\) such that \(\text{is(m, *p)}\) would return \text{true}; otherwise, returns \text{high}.

\begin{verbatim}
const char* do_scan_not(mask m, const charT* low, const charT* high) const;
\end{verbatim}

Effects: Locates a character in a buffer that fails to conform to a classification \text{m}.

Returns: The smallest pointer \(p\), if any, in the range \([\text{low}, \text{high})\) such that \(\text{is(m, *p)}\) would return \text{false}; otherwise, returns \text{high}.

\begin{verbatim}
charT do_toupper(charT c) const;
\end{verbatim}

\begin{verbatim}
const charT* do_toupper(charT* low, const charT* high) const;
\end{verbatim}

Effects: Converts a character or characters to upper case. The second form replaces each character \(*p\) in the range \([\text{low, high})\) 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 \text{high}.

\begin{verbatim}
charT do_tolower(charT c) const;
\end{verbatim}

\begin{verbatim}
const charT* do_tolower(charT* low, const charT* high) const;
\end{verbatim}

Effects: Converts a character or characters to lower case. The second form replaces each character \(*p\) in \([\text{low, high})\) and for which a corresponding lower-case character exists, with that character.

Returns: The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns \text{high}.

\begin{verbatim}
charT do_widen(char c) const;
\end{verbatim}

\begin{verbatim}
const charT* do_widen(const charT* low, const charT* high, charT* dest) const;
\end{verbatim}

Effects: Applies the simplest reasonable transformation from a \text{char} value or sequence of \text{char} values to the corresponding \text{charT} value or values.\textsuperscript{264} The only characters for which unique transformations are required are those in the basic source character set (5.3).

For any named \text{ctype} category with a \text{ctype <charT>} facet \text{ctc} and valid \text{ctype_base::mask} value \(M\),
\[(\text{ctc.is(M, c) \lor \text{!is(M, do_widen(c)))} \Rightarrow \text{true})\]

The second form transforms each character \(*p\) in the range \([\text{low, high})\), placing the result in \(\text{dest}[p - \text{low}]\).

Returns: The first form returns the transformed value. The second form returns \text{high}.

\begin{verbatim}
char do_narrow(charT c, char dfault) const;
\end{verbatim}

\begin{verbatim}
const charT* do_narrow(const charT* low, const charT* high, char dfault, charT* dest) const;
\end{verbatim}

Effects: Applies the simplest reasonable transformation from a \text{charT} value or sequence of \text{charT} values to the corresponding \text{char} value or values.

For any character \(c\) in the basic source character set (5.3) the transformation is such that
\[
\text{do_widen(do_narrow}(c, 0)) == \text{c}
\]

For any named \text{ctype} category with a \text{ctype<char>} facet \text{ctc} however, and \text{ctype_base::mask} value \(M\),
\[
(\text{is(M, c) \lor !\text{ctc.is(M, do_narrow(c, dfault)))})\]
is \text{true} (unless \text{do_narrow} returns \text{dfault}). In addition, for any digit character \(c\), the expression \((\text{do_-narrow}(c, \text{dfault}) - '0')\) evaluates to the digit value of the character. The second form transforms each character \(*p\) in the range \([\text{low, high})\), placing the result (or \text{dfault} if no simple transformation is readily available) in \(\text{dest}[p - \text{low}]\).

\textsuperscript{264} The \text{char} argument of \text{do_widen} is intended to accept values derived from character literals for conversion to the locale’s encoding.

\textsuperscript{265} In other words, the transformed character is not a member of any character classification that \(c\) is not also a member of.
Returns: The first form returns the transformed value; or default 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> {
        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();
    };
}

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) 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;

        protected:
            ~ctype();
            virtual char do_toupper(char c) const;
            virtual const char* do_toupper(char* low, const char* high) const;
            virtual char do_tolower(char c) const;
            virtual const char* do_tolower(char* low, const char* high) const;
            virtual char do_widen(char c) const;
            virtual const char* do_widen(const char* low, const char* high, char* to) const;
            virtual char do_narrow(char c, char dfault) const;
            virtual const char* do_narrow(const char* low, const char* high, char dfault, char* to) const;
    };
}
A specialization \texttt{ctype\textless char\textgreater} is provided so that the member functions on type \texttt{char} can be implemented inline.\footnote{Only the \texttt{char} (not \texttt{unsigned char} and \texttt{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 \texttt{ctype\textless char\textgreater}.} The implementation-defined value of member \texttt{table\_size} is at least 256.

28.4.1.3.1 Destructor

\texttt{~ctype();}

\textbf{Effects:} If the constructor’s first argument was nonzero, and its second argument was \texttt{true}, does delete [] table().

28.4.1.3.2 Members

In the following member descriptions, for \texttt{unsigned char} values \(v\) where \(v >= \text{table\_size}\), \texttt{table()[v]} is assumed to have an implementation-specific value (possibly different for each such value \(v\)) without performing the array lookup.

\texttt{explicit ctype(const mask* tbl = nullptr, bool del = false, size\_t refs = 0);}

\textbf{Requires:} tbl either 0 or an array of at least \texttt{table\_size} elements.

\textbf{Effects:} Passes its \texttt{refs} argument to its base class constructor.

\begin{enumerate}
\item \texttt{bool is(mask m, char c) const;}
\item \texttt{const char* is(const char* low, const char* high, mask* vec) const;}
\item \texttt{bool is(mask m, char c) const;}
\item \texttt{const char* is(const char* low, const char* high, mask* vec) const;}
\item \texttt{const char* scan_is(mask m, const char* low, const char* high) const;}
\item \texttt{const char* scan_not(mask m, const char* low, const char* high) const;}
\item \texttt{char toupper(char c) const;}
\item \texttt{char tolower(char c) const;}
\item \texttt{char widen(char c) const;}
\item \texttt{char narrow(char c, char dfault) const;}
\item \texttt{const mask* table() const noexcept;}
\end{enumerate}

\textbf{Returns:} The first form returns \texttt{table()[(unsigned char)c & m}}; the second form returns \texttt{high}.

\textbf{Returns:} The smallest \(p\) in the range \([low, high)\) such that \(table()[(unsigned char)*p] & m\) is \texttt{true}.

\textbf{Returns:} The smallest \(p\) in the range \([low, high)\) such that \(table()[(unsigned char)*p] & m\) is \texttt{false}.

\textbf{Returns:} \texttt{do\_toupper(c)} or \texttt{do\_toupper(low, high)}, respectively.

\textbf{Returns:} \texttt{do\_tolower(c)} or \texttt{do\_tolower(low, high)}, respectively.

\textbf{Returns:} \texttt{do\_widen(c)} or \texttt{do\_widen(low, high, to)}, respectively.

\textbf{Returns:} \texttt{do\_narrow(c)} or \texttt{do\_narrow(low, high, dfault, to)}, respectively.

\textbf{Returns:} The first constructor argument, if it was nonzero, otherwise \texttt{classic\_table()}.\footnote{Only the \texttt{char} (not \texttt{unsigned char} and \texttt{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 \texttt{ctype\textless char\textgreater}.}
28.4.1.3.3 Static members

static const mask* classic_table() noexcept;

1 Returns: A pointer to the initial element of an array of size table_size which represents the classifications of characters in the "C" locale.

28.4.1.3.4 Virtual functions

cchar do_toupper(char) const;
cconst char* do_toupper(char* low, const char* high) const;
cchar do_tolower(char) const;
cconst char* do_tolower(char* low, const char* high) const;

virtual cchar do_widen(char c) const;
virtual const char* do_widen(char* low, const char* high, char* to) const;
virtual cchar do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low, const char* high,
char dfault, char* to) const;

1 These functions are described identically as those members of the same name in the ctype class template (28.4.1.1.1).

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;

§ 28.4.1.4
virtual result do_in(
    stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to,     internT* to_end,       internT*& to_next) const;
virtual result do_unshift(
    stateT& state,
    externT* to,     externT* to_end,       externT*& to_next) const;

virtual int do_encoding() const noexcept;
virtual bool do_always_noconv() const noexcept;
virtual int do_length(stateT& state, const externT* from, const externT* end, size_t max) const;
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 Unicode and EUC.

2 The stateT argument selects the pair of character encodings being mapped between.

3 The specializations required in Table 90 (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

result out(
    stateT& state,
    const internT* from, const internT* from_end, const internT*& from_next,
    externT* to,     externT* to_end,       externT*& to_next) const;
1 Returns: do_out(state, from, from_end, from_next, to, to_end, to_next).

result unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;
2 Returns: do_unshift(stateT& state, externT* to, externT* to_end, externT*& to_next).

result in(
    stateT& state,
    const externT* from, const externT* from_end, const externT*& from_next,
    internT* to,     internT* to_end,       internT*& to_next) const;
3 Returns: do_in(stateT& state, externT* from, externT* from_end, externT*& from_next).

int encoding() const noexcept;
4 Returns: do_encoding().

bool always_noconv() const noexcept;
5 Returns: do_always_noconv().

int length(stateT& state, const externT* from, const externT* from_end, size_t max) const;
6 Returns: do_length(stateT& state, externT* from, externT* from_end, max).

int max_length() const noexcept;
7 Returns: do_max_length().
28.4.1.4.2 Virtual functions

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;

1 Requires: (from <= from_end && to <= to_end) well-defined and true: state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

2 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). 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).

3 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.267 [Note: As a result of operations on state, it can return ok or partial and set from_next == from and to_next != to. — end note]

4 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]

5 Returns: An enumeration value, as summarized in Table 92.

<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) 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.

267 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.
result do_unshift(stateT& state, externT* to, externT* to_end, externT*& to_next) const;

Requires: (to <= to_end) well-defined and true; state initialized, if at the beginning of a sequence, or else 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. 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 93.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ok</td>
<td>completed the sequence</td>
</tr>
<tr>
<td>partial</td>
<td>space for more than to_end - to destination elements was needed to terminate a sequence given the value of state</td>
</tr>
<tr>
<td>error</td>
<td>an unspecified error has occurred</td>
</tr>
<tr>
<td>noconv</td>
<td>no termination is needed for this state_type</td>
</tr>
</tbody>
</table>

int do_encoding() const noexcept;

Returns: -1 if the encoding of the externT sequence is state-dependent; else the constant number of externT characters needed to produce an internal character; or 0 if this number is not a constant.

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;

Requires: (from <= from_end) well-defined and true; state initialized, if at the beginning of a sequence, or else 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 codecvt_byname

namespace std {
    template<class internT, class externT, class stateT>
    class codecvt_byname : public codecvt<internT, externT, stateT> {
    public:
        explicit codecvt_byname(const char*, size_t refs = 0);
        explicit codecvt_byname(const string&, size_t refs = 0);

    protected:
        ~codecvt_byname();
    }
}

268) Typically these will be characters to return the state to stateT().
269) 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.
28.4.2 The numeric category

1 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.270

2 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 90 and 91 (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 numpunct<> facet to identify all numeric punctuation preferences, and also for the ctype<> facet to perform character classification.

3 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, 29.7.5.2.1).

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;

270) Parsing "-1" correctly into, e.g., an unsigned short requires that the corresponding member get() at least extract the sign before delegating.
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

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

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;

Effects: Reads characters from in, interpreting them according to str.flags(), use_facet<ctype<
cartment<charT>>(loc), and use_facet<numpunct<charT>>(loc), where loc is str.getloc().

The details of this operation occur in three stages

Stage 1: Determine a conversion specifier
Stage 2: Extract characters from in and determine a corresponding char value for the format
expected by the conversion specification determined in stage 1.
Stage 3: Store results

The details of the stages are presented below.

Stage 1: The function initializes local variables via
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 94. The table is ordered. That is, the first line whose condition is true applies.

Table 94 — Integer conversions

<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 95.

Table 95 — Length modifier

<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
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:
static const char src[] = "0123456789abcdefxABCDEFX+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT>>(loc).widen(src, src + sizeof(src), atoms);
for this value of \texttt{loc}.

If \texttt{discard} is \texttt{true}, then if \texttt{'}.'\ has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if \texttt{'}.'\ 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 \texttt{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 \texttt{in} is advanced by ++\texttt{in} and processing returns to the beginning of stage 2.

\textbf{Stage 3:} The sequence of \texttt{chars} accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header \texttt{<cstdlib>}: 

\begin{enumerate}
\item For a signed integer value, the function \texttt{strtol}.
\item For an unsigned integer value, the function \texttt{strtoul}.
\item For a \texttt{float} value, the function \texttt{strtof}.
\item For a \texttt{double} value, the function \texttt{strtod}.
\item For a \texttt{long double} value, the function \texttt{strtold}.
\end{enumerate}

The numeric value to be stored can be one of:

\begin{enumerate}
\item zero, if the conversion function does not convert the entire field.
\item 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 \texttt{val}.
\item the most positive representable value, if the field to be converted to an unsigned integer type represents a value that cannot be represented in \texttt{val}.
\item the converted value, otherwise.
\end{enumerate}

The resultant numeric value is stored in \texttt{val}. If the conversion function does not convert the entire field, or if the field represents a value outside the range of representable values, \texttt{ios\_base::failbit} is assigned to \texttt{err}.

Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with \texttt{use\_facet<numpunct\textless charT\textgreater \texttt{(loc).grouping()}}. If they are not consistent then \texttt{ios\_base::failbit} is assigned to \texttt{err}.

In any case, if stage 2 processing was terminated by the test for \texttt{in == end} then \texttt{err |= ios\_base::eofbit} is performed.

```cpp
iter_type do_get(iter_type in, iter_type end, ios_base& str, 
                 ios_base::iostate& err, bool& val) const;
```

\textbf{Effects:} If \texttt{(str.flags()&ios\_base::boolalpha)} == 0 then input proceeds as it would for a \texttt{long} except that if a value is being stored into \texttt{val}, the value is determined according to the following: If the value to be stored is 0 then \texttt{false} is stored. If the value is 1 then \texttt{true} is stored. Otherwise \texttt{true} is stored and \texttt{ios\_base::failbit} is assigned to \texttt{err}.

Otherwise target sequences are determined “as if” by calling the members \texttt{false\_name()} and \texttt{true\_name()} of the facet obtained by \texttt{use\_facet<numpunct\textless charT\textgreater \texttt{(str.getloc())}}. Successive characters in the range \texttt{[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 \texttt{in} is compared to \texttt{end} only when necessary to obtain a character. If a target sequence is uniquely matched, \texttt{val} is set to the corresponding value. Otherwise \texttt{false} is stored and \texttt{ios\_base::failbit} is assigned to \texttt{err}.

The \texttt{in} iterator is always left pointing one position beyond the last character successfully matched. If \texttt{val} is set, then \texttt{err} is set to \texttt{str\_goodbit}; or to \texttt{str\_eofbit} if, when seeking another character to match, it is found that (\texttt{in == end}). If \texttt{val} is not set, then \texttt{err} is set to \texttt{str\_failbit}; or to (\texttt{str\_failbit}|\texttt{str\_eofbit}) if the reason for the failure was that (\texttt{in == end}). [Example: For targets \texttt{true: "a" and false: "abb"}, the input sequence \texttt{"a"} yields \texttt{val == true} and \texttt{err == str\_eofbit}; the input sequence \texttt{"abc"} yields \texttt{err == str\_failbit}, with \texttt{in} ending at the \texttt{'}c'} element. For targets \texttt{true: "1" and false: "0"}, the input sequence \texttt{"1"} yields \texttt{val == true} and \texttt{err == str\_goodbit}. For empty targets (\texttt{""}), any input sequence yields \texttt{err == str\_failbit}. — end example]

\textbf{Returns:} \texttt{in}. 

§ 28.4.2.1.2

1291
28.4.2.2 Class template `num_put`

namespace std {
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class num_put : public locale::facet {
        public:
            using char_type = charT;
            using iter_type = OutputIterator;
            explicit num_put(size_t refs = 0);
            iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long long v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
            iter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;
        protected:
            ~num_put();
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, bool v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, long v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, long long v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, unsigned long long v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, double v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, long double v) const;
            virtual iter_type do_put(iter_type s, ios_base& f, char_type fill, const void* v) const;
    };

The facet `num_put` is used to format numeric values to a character sequence such as an ostream.

28.4.2.2.1 Members

```
iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, unsigned long long v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, double v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, long double v) const;
iter_type put(iter_type s, ios_base& f, char_type fill, const void* v) const;
```

Returns: `do_put(out, str, fill, val)`.

28.4.2.2.2 Virtual functions

```
iter_type do_put(iter_type s, ios_base& f, char_type fill, long v) const;
iter_type do_put(iter_type s, ios_base& f, char_type fill, long long v) const;
iter_type do_put(iter_type s, ios_base& f, char_type fill, unsigned long v) const;
iter_type do_put(iter_type s, ios_base& f, char_type fill, unsigned long long v) const;
iter_type do_put(iter_type s, ios_base& f, char_type fill, double v) const;
iter_type do_put(iter_type s, ios_base& f, char_type fill, long double v) const;
iter_type do_put(iter_type s, ios_base& f, char_type fill, const void* v) const;
```

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.

— 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)

— Stage 3: Determine where padding is required.

— Stage 4: Insert the sequence into the out.

Detailed descriptions of each stage follow.

Returns: out.

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

```c
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 96.

**Table 96 — Integer conversions**

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == ios_base::oct</td>
<td>%o</td>
</tr>
<tr>
<td>(basefield == ios_base::hex) &amp;&amp; !uppercase</td>
<td>%x</td>
</tr>
<tr>
<td>(basefield == ios_base::hex)</td>
<td>%X</td>
</tr>
<tr>
<td>for a signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>for an unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 97.

**Table 97 — Floating-point conversions**

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatfield == ios_base::fixed</td>
<td>%f</td>
</tr>
<tr>
<td>floatfield == ios_base::scientific &amp;&amp; !uppercase</td>
<td>%e</td>
</tr>
<tr>
<td>floatfield == ios_base::scientific</td>
<td>%E</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>%g</td>
</tr>
<tr>
<td>otherwise</td>
<td>%G</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 98.

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 99.

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.
Table 98 — Length modifier

<table>
<thead>
<tr>
<th>Type</th>
<th>Length modifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>l</td>
</tr>
<tr>
<td>long long</td>
<td>ll</td>
</tr>
<tr>
<td>unsigned long</td>
<td>l</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>

Table 99 — Numeric conversions

<table>
<thead>
<tr>
<th>Type(s) State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>an integral type</td>
<td>showpos +</td>
</tr>
<tr>
<td></td>
<td>showbase #</td>
</tr>
<tr>
<td>a floating-point type</td>
<td>showpos +</td>
</tr>
<tr>
<td></td>
<td>showpoint #</td>
</tr>
</tbody>
</table>

The representations at the end of stage 1 consists of the char’s that would be printed by a call of printf(s, val) where s is the conversion specifier determined above.

Stage 2: Any character c other than a decimal point(.) is converted to a charT via

```
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\(^271\) is determined according to Table 100.

<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 0 or 0X</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

\(^{271}\) The conversion specification #o generates a leading 0 which is not a padding character.
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

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;

      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 90 (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:

integer ::= [sign] units
sign ::= plusminus
plusminus ::= '++' | '--'
units ::= digits [thousands-sep units]
digits ::= digit [digits]

and floating-point values have:

floatval ::= [sign] units [decimal-point [digits]] [e [sign] digits] |
  [sign] decimal-point digits [e [sign] 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

char_type decimal_point() const;
  Returns: do_decimal_point().

char_type thousands_sep() const;
  Returns: do_thousands_sep().

§ 28.4.3.1.1
string grouping() const;
3    Returns: do_grouping().

string_type truename() const;
string_type falsename() const;
4    Returns: do_truename() or do_falsename(), respectively.

28.4.3.1.2 Virtual functions [facet.numpunct.virtuals]

char_type do_decimal_point() const;
1    Returns: A character for use as the decimal radix separator. The required specializations return "." or L"."

char_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 basic_string<char> vec used as a vector of integer values, in which each element vec[i]
represents the number of digits\textsuperscript{272} in the group at position i, starting with position 0 as the rightmost
group. If vec.size() \(\leq\) i, the number is the same as group \((i - 1)\); if \((i < 0 \| vec[i] \leq 0
\| vec[i] ==\text{CHAR}\_\text{MAX})\), the size of the digit group is unlimited.
4    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 \texttt{true} or \texttt{false}, respectively.
6    In the base class implementation these names are \texttt{"true"} and \texttt{"false"}, or L\texttt{"true"} and L\texttt{"false"}.

28.4.3.2 Class template numpunct\_byname [locale.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 [category.collate]

28.4.4.1 Class template collate [locale.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);

\textsuperscript{272} Thus, the string \texttt{"\003"} specifies groups of 3 digits each, and \texttt{"3"} probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of \texttt{"3"}.
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 90 (28.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.7.10).

Each function compares a string of characters `*p` in the range `[low, high)`.

### 28.4.4.1.1 Members

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>28.4.4.1.1 Members</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;</code></td>
<td>Returns: <code>do_compare(low1, high1, low2, high2)</code></td>
<td>locale.collate.members</td>
</tr>
<tr>
<td><code>string_type transform(const charT* low, const charT* high) const;</code></td>
<td>Returns: A <code>basic_string&lt;charT&gt;</code> value that, compared lexicographically with the result of calling <code>transform()</code> on another string, yields the same result as calling <code>do_compare()</code> on the same two strings.273</td>
<td>locale.collate.members</td>
</tr>
<tr>
<td><code>long hash(const charT* low, const charT* high) const;</code></td>
<td>Returns: An integer value equal to the result of calling <code>hash()</code> on any other string for which <code>do_compare()</code> returns 0 (equal) when passed the two strings. <strong>[Note:</strong> The probability that the result equals that for another string which does not compare equal should be very small, approaching (1.0/numeric_limits&lt;unsigned long&gt;::max()). — end note]**</td>
<td>locale.collate.members</td>
</tr>
</tbody>
</table>

### 28.4.4.1.2 Virtual functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>28.4.4.1.2 Virtual functions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int do_compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;</code></td>
<td>Returns: 1 if the first string is greater than the second, -1 if less, zero otherwise. The specializations required in Table 90 (28.3.1.1.1), namely <code>collate&lt;char&gt;</code> and <code>collate&lt;wchar_t&gt;</code>, implement a lexicographical comparison (25.7.10).</td>
<td>locale.collate.virtuals</td>
</tr>
<tr>
<td><code>string_type do_transform(const charT* low, const charT* high) const;</code></td>
<td>Returns: A <code>basic_string&lt;charT&gt;</code> value that, compared lexicographically with the result of calling <code>transform()</code> on another string, yields the same result as calling <code>do_compare()</code> on the same two strings.273</td>
<td>locale.collate.virtuals</td>
</tr>
<tr>
<td><code>long do_hash(const charT* low, const charT* high) const;</code></td>
<td>Returns: An integer value equal to the result of calling <code>hash()</code> on any other string for which <code>do_compare()</code> returns 0 (equal) when passed the two strings. <strong>[Note:</strong> The probability that the result equals that for another string which does not compare equal should be very small, approaching (1.0/numeric_limits&lt;unsigned long&gt;::max()). — end note]**</td>
<td>locale.collate.virtuals</td>
</tr>
</tbody>
</table>

### 28.4.4.2 Class template `collatebyname`

```cpp
namespace std {
    template<class charT>
    class collatebyname : public collate<charT> {
        public:
            using string_type = basic_string<charT>;
    };
}
```

273) This function is useful when one string is being compared to many other strings.
explicit collate_byname(const char*, size_t refs = 0);
explicit collate_byname(const string&, size_t refs = 0);

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 90 and 91 (28.3.1.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in 28.4, and the ctype<> facet, to determine formatting details.

28.4.5.1 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,
        ios_base::iostate& err, tm* t, char format, char modifier = 0) const;
    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;

    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& f,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_date(iter_type s, iter_type end, ios_base& f,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base& f,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base& f,
        ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_year(iter_type s, iter_type end, ios_base& f,
        ios_base::iostate& err, tm* t) const;
};

§ 28.4.5.1 1298
virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm* t, char format, char modifier) const;
};

1 time_get is used to parse a character sequence, extracting components of a time or date into a struct tm object. Each get member parses a format as produced by a corresponding format specifier to time_put<>::put. If the sequence being parsed matches the correct format, the corresponding members of the struct tm argument are set to the values used to produce the sequence; otherwise either an error is reported or unspecified values are assigned.274

2 If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_base::eofbit in err.

28.4.5.1.1 Members

dateorder date_order() const;  
Returns: do_date_order().

iter_type get_time(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
Returns: do_get_time(s, end, str, err, t).

iter_type get_date(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
Returns: do_get_date(s, end, str, err, t).

iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
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).

iter_type get_year(iter_type s, iter_type end, ios_base& str,
    ios_base::iostate& err, tm* t) const;
Returns: do_get_year(s, end, str, err, t).

iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
    tm* t, char format, char modifier = 0) const;
Returns: do_get(s, end, f, err, t, format, modifier).

iter_type get(iter_type s, iter_type end, ios_base& f, ios_base::iostate& err,
    tm* t, const char_type* fmt, const char_type* fmtend) const;
Requires: [fmt, fmtend] shall be 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:

(8.1) The expression fmt == fmtend evaluates to true.
(8.2) The expression err == ios_base::goodbit evaluates to false.
(8.3) The expression s == end evaluates to true, in which case the function evaluates err = ios_base::eofbit | ios_base::failbit.
(8.4) 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 strftime. 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

274) 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.
the optional modifier is absent from the conversion specification. If \( \text{err} == \text{ios}_\text{base}::\text{goodbit} \) holds after the evaluation of the expression, the function increments \( \text{fmt} \) to point just past the end of the conversion specification and continues looping.

— The expression \( \text{isspace}(*\text{fmt}, \text{f.getloc}()) \) evaluates to \( \text{true} \), in which case the function first increments \( \text{fmt} \) until \( \text{fmt} == \text{fmtend} || !\text{isspace}(*\text{fmt}, \text{f.getloc}()) \) evaluates to \( \text{true} \), then advances \( \text{s} \) until \( \text{s} == \text{end} || !\text{isspace}(*\text{s}, \text{f.getloc}()) \) is \( \text{true} \), and finally resumes looping. Otherwise, the function evaluates \( \text{err} = \text{ios}_\text{base}::\text{failbit} \).

— (8.5)

The next character read from \( \text{s} \) matches the element pointed to by \( \text{fmt} \) in a case-insensitive comparison, in which case the function first increments \( \text{fmt} \) until \( \text{fmt} == \text{fmtend} \) or \( \text{isspace}(*\text{fmt}, \text{f.getloc}()) \) is \( \text{false} \), then advances \( \text{s} \) until \( \text{s} == \text{end} \) or \( \text{isspace}(*\text{s}, \text{f.getloc}()) \) is \( \text{false} \), and finally resumes looping. Otherwise, the function evaluates \( \text{err} = \text{ios}_\text{base}::\text{failbit} \).

— (8.6)

[Note: The function uses the \text{ctype<charT>} facet installed in \( \text{f} \)'s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so. — end note]

Returns: \( \text{s} \).

28.4.5.1.2 Virtual functions

\text{dateorder do_date_order() const;}

1

Returns: An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.\( ^{275} \) Returns \text{no_order} if the date format specified by 'x' contains other variable components (e.g., Julian day, week number, week day).

\text{iter_type do_get_time(iter_type s, iter_type end, ios_base& str,}
\text{ios_base::iostate& err, tm* t) const;}

2

Effects: Reads characters starting at \( \text{s} \) until it has extracted those \text{struct tm} members, and remaining format characters, used by \text{time_put<>::put} to produce the format specified by ‘%H:%M:%S’, or until it encounters an error or end of sequence.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

\text{iter_type do_get_date(iter_type s, iter_type end, ios_base& str,}
\text{ios_base::iostate& err, tm* t) const;}

3

Effects: Reads characters starting at \( \text{s} \) until it has extracted those \text{struct tm} members and remaining format characters used by \text{time_put<>::put} to produce one of the following formats, or until it encounters an error. The format depends on the value returned by \text{date_order()} as shown in Table 101.

Table 101 — \text{do_get_date} effects

<table>
<thead>
<tr>
<th>date_order()</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_order</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>dmy</td>
<td>&quot;%d%m%y&quot;</td>
</tr>
<tr>
<td>mdy</td>
<td>&quot;%m%d%y&quot;</td>
</tr>
<tr>
<td>ymd</td>
<td>&quot;%y%m%d&quot;</td>
</tr>
<tr>
<td>ydm</td>
<td>&quot;%y%d%m&quot;</td>
</tr>
</tbody>
</table>

5 An implementation may also accept additional implementation-defined formats.

6 Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

\text{iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str,}
\text{ios_base::iostate& err, tm* t) const;}

7 Effects: Reads characters starting at \( \text{s} \) 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

\footnote{275 This function is intended as a convenience only, for common formats, and may return \text{no_order} in valid locales.}
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.

```cpp
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.

```cpp
iter_type do_get(iter_type s, iter_type end, ios_base& f,  
                 ios_base::iostate& err, tm* t, char format, char modifier) const;
```

**Requires:** `t` shall point to an object.

**Effects:** The function starts by evaluating `err = ios_base::goodbit`. It then reads characters starting at `s` until it encounters an error, or until it has extracted and assigned those `struct tm` members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function `strptime`, formed by concatenating '%', the `modifier` character, when non-NUL, and the `format` character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by `t` unchanged and evaluates `err |= ios_base::failbit`. When `s == end` evaluates to `true` after reading a character the function evaluates `err |= ios_base::eofbit`.

For complex conversion directives such as `%c`, `%x`, or `%X`, or directives that involve the optional modifiers `E` or `O`, when the function is unable to unambiguously determine some or all `struct tm` members from the input sequence `[s, end)`, it evaluates `err |= ios_base::eofbit`. In such cases the values of those `struct tm` members are unspecified and may be outside their valid range.

**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_getbyname

```cpp
namespace std {
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_getbyname : public time_get<charT, InputIterator> {
        public:
            using dateorder = time_base::dateorder;
            using iter_type = InputIterator;

            explicit time_getbyname(const char*, size_t refs = 0);
            explicit time_getbyname(const string&, size_t refs = 0);

            protected:
                ~time_getbyname();
        };
}
```

### 28.4.5.3 Class template time_put

```cpp
namespace std {
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put : public locale::facet {
        public:
            using char_type = charT;
            using iter_type = OutputIterator;

            explicit time_put(size_t refs = 0);
    }
}
```
// the following is implemented in terms of other member functions.
iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& f, char_type fill,
const tm* tmb, char format, char modifier = 0) const;

static locale::id id;

protected:
~time_put();
virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t,
char format, char modifier) const;
};

28.4.5.3.1 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 mod276 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&, 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()277, except that the
sequence of characters produced for those specifiers that are described as depending on the C locale are
instead implementation-defined.278

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;

276) Although the C programming language defines no modifiers, most vendors do.
277) Interpretation of the modifier argument is implementation-defined, but should follow POSIX conventions.
278) Implementations should refer to other standards such as POSIX for these definitions.
explicit time_put_byname(const char*, size_t refs = 0);
explicit time_put_byname(const string&, size_t refs = 0);

protected:
~time_put_byname();
};

28.4.6 The monetary category

These templates handle monetary formats. A template parameter indicates whether local or international
monetary formats are to be used.

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 90 and 91 (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;
    }

    static locale::id id;
}

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;

1 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;

1 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 \texttt{err}; otherwise, sets \texttt{err} to \texttt{(err|str.failbit)}, or \texttt{(err|str.failbit|str.eofbit)} if no more characters are available, and does not change \texttt{units} or \texttt{digits}. Uses the pattern returned by \texttt{mp.neg_format()} to parse all values. The result is returned as an integral value stored in \texttt{units} or as a sequence of digits possibly preceded by a minus sign (as produced by \texttt{ct.widen(c)} where \texttt{c} is \texttt{'-'}, or in the range from \texttt{'0'} through \texttt{'9'} (inclusive)) stored in \texttt{digits}. [Example: The sequence \texttt{$1,056.23$} in a common United States locale would yield, for \texttt{units}, \texttt{105623}, or, for digits, \texttt{"105623"}. — end example] If \texttt{mp.grouping()} indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

Where \texttt{money_base::space} or \texttt{money_base::none} appears as the last element in the format pattern, no white space is consumed. Otherwise, where \texttt{money_base::space} appears in any of the initial elements of the format pattern, at least one white space character is required. Where \texttt{money_base::none} appears in any of the initial elements of the format pattern, white space is allowed but not required. If \texttt{(str.flags() \& str.showbase)} is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

If the first character (if any) in the string \texttt{pos} returned by \texttt{mp.positive_sign()} or the string \texttt{neg} returned by \texttt{mp.negative_sign()} is recognized in the position indicated by \texttt{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 \texttt{showbase} is off, then for a \texttt{neg} value of \texttt{"()"} and a currency symbol of \texttt{"L"}, in \texttt{"(100 L)"} the \texttt{"L"} is consumed; but if \texttt{neg} is \texttt{"-"}, the \texttt{"L"} in \texttt{"-100 L"} is not consumed. — end example] If \texttt{pos} or \texttt{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 \texttt{pos} or \texttt{neg}, and the result is given the corresponding sign. If the first character of \texttt{pos} is equal to the first character of \texttt{neg}, or if both strings are empty, the result is given a positive sign.

Digits in the numeric monetary component are extracted and placed in \texttt{digits}, or into a character buffer \texttt{buf1} for conversion to produce a value for \texttt{units}, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value \texttt{units} is produced as if by

\begin{verbatim}
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);
\end{verbatim}

where \texttt{n} is the number of characters placed in \texttt{buf1}, \texttt{buf2} is a character buffer, and the values \texttt{src} and \texttt{atoms} are defined as if by

\begin{verbatim}
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
\end{verbatim}

\texttt{Returns:} An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

\section*{28.4.6.2 Class template \texttt{money_put}}

\begin{verbatim}
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;
  };
\end{verbatim}

\footnote{279) The semantics here are different from \texttt{ct.narrow}.}
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;
1 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;
1 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]
3 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);

§ 28.4.6.3
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:

```plaintext
value ::= units [ decimal-point [ digits ]] |
   decimal-point digits
```

if `frac_digits()` returns a positive value, or

```plaintext
value ::= units
```

otherwise. The symbol `decimal-point` indicates the character returned by `decimal_point()`. The other symbols are defined as follows:

```plaintext
units ::= digits [ thousands-sep units ]
digits ::= adigit [ digits ]
```

In the syntax specification, the symbol `adigit` is any of the values `ct.widen(c)` for `c` in the range '0' through '9' (inclusive) and `ct` is a reference of type `const ctype<charT>&` obtained as described in the definitions of `money_get<>` and `money_put<>`. The symbol `thousands-sep` is the character returned by `thousands_sep()`. The space character used is the value `ct.widen(' ')`. White space characters are those characters `c` for which `ci.is(space, c)` returns `true`. The number of digits required after the decimal point (if any) is exactly the value returned by `frac_digits()`.

The placement of thousands-separator characters (if any) is determined by the value returned by `grouping()`, defined identically as the member `numpunct<>::do_grouping()`.

---

280) An array of `char`, rather than an array of `part`, is specified for `pattern::field` purely for efficiency.
28.4.6.3.1 Members

<table>
<thead>
<tr>
<th>charT</th>
<th>decimal_point() const;</th>
</tr>
</thead>
<tbody>
<tr>
<td>charT</td>
<td>thousands_sep() const;</td>
</tr>
<tr>
<td>string</td>
<td>grouping() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>curr_symbol() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>positive_sign() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>negative_sign() const;</td>
</tr>
<tr>
<td>int</td>
<td>frac_digits() const;</td>
</tr>
<tr>
<td>pattern</td>
<td>pos_format() const;</td>
</tr>
<tr>
<td>pattern</td>
<td>neg_format() const;</td>
</tr>
</tbody>
</table>

1 Each of these functions \( F \) returns the result of calling the corresponding virtual member function \( \text{do}_F() \).

28.4.6.3.2 Virtual functions

<table>
<thead>
<tr>
<th>charT</th>
<th>do_decimal_point() const;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns: The radix separator to use in case ( \text{do_frac_digits}() ) is greater than zero. ( ^{281} )</td>
<td></td>
</tr>
<tr>
<td>charT</td>
<td>do_thousands_sep() const;</td>
</tr>
<tr>
<td>Returns: The digit group separator to use in case ( \text{do_grouping}() ) specifies a digit grouping pattern. ( ^{282} )</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>do_grouping() const;</td>
</tr>
<tr>
<td>Returns: A pattern defined identically as, but not necessarily equal to, the result of ( \text{numpunct&lt;\charT&gt;::do_grouping()} ). ( ^{283} )</td>
<td></td>
</tr>
<tr>
<td>string_type</td>
<td>do_curr_symbol() const;</td>
</tr>
<tr>
<td>Returns: A string to use as the currency identifier symbol. ( \text{Note: For specializations where the second template parameter is } \text{true}, \text{this is typically four characters long: a three-letter code as specified by ISO 4217 followed by a space.} ) — end note</td>
<td></td>
</tr>
<tr>
<td>string_type</td>
<td>do_positive_sign() const;</td>
</tr>
<tr>
<td>string_type</td>
<td>do_negative_sign() const;</td>
</tr>
<tr>
<td>Returns: ( \text{do_positive_sign()} ) returns the string to use to indicate a positive monetary value; ( ^{284} ) ( \text{do_negative_sign()} ) returns the string to use to indicate a negative value.</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>do_frac_digits() const;</td>
</tr>
<tr>
<td>Returns: The number of digits after the decimal radix separator, if any. ( ^{285} )</td>
<td></td>
</tr>
<tr>
<td>pattern</td>
<td>do_pos_format() const;</td>
</tr>
<tr>
<td>pattern</td>
<td>do_neg_format() const;</td>
</tr>
<tr>
<td>Returns: The specializations required in Table 91 (28.3.1.1.1), namely ( \text{moneypunct&lt;\charT&gt;, moneypunct&lt;wchar_t&gt;, moneypunct&lt;\charT, true&gt;, and moneypunct&lt;wchar_t, true&gt;} ), return an object of type ( \text{pattern} ) initialized to { \text{symbol, sign, none, value} }. ( ^{286} )</td>
<td></td>
</tr>
</tbody>
</table>

28.4.6.4 Class template moneypunct_byname

```cpp
class moneypunct_byname : public moneypunct<\charT, \text{true}> { 
  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);
}
```

281) In common U.S. locales this is \'\,'.
282) In common U.S. locales this is \'\',\.'
283) To specify grouping by 3s, the value is \\"\003\" \text{not \"3\"}.
284) This is usually the empty string.
285) In common U.S. locales, this is 2.
286) Note that the international symbol returned by \( \text{do_curr_symbol()} \) usually contains a space, itself; for example, \\"USD \."
28.4.7 The message retrieval category
1 Class messages<
Class messages<
Class messages<charT> implements retrieval of strings from message catalogs.

28.4.7.1 Class template messages
namespace std {

namespace std {

    class messages_base {
        using catalog = unspecified signed integer type;
    };

    template<class charT>
    class messages : public locale::facet, public messages_base {
        using char_type = charT;
        using string_type = basic_string<charT>;

        explicit messages(size_t refs = 0);
        catalog open(const basic_string<char>& fn, const locale&) const;
        string_type get(catalog cat, int set, int msgid, const string_type& dfault) const;
        void close(catalog cat) const;

        static locale::id id;
    };

protected:
    ~messages();
    virtual catalog do_open(const basic_string<char>& name, const locale&) const;
    virtual string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const;
    virtual void do_close(catalog cat) 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 basic_string<char>& name, const locale& loc) const;
1 Returns: do_open(name, loc).

string_type get(catalog cat, int set, int msgid, 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

catalog do_open(const basic_string<char>& 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().

Returns a value less than 0 if no such catalog can be opened.

Remarks: The locale argument loc is used for character set code conversion when retrieving messages,
if needed.
string_type do_get(catalog cat, int set, intmsgid, const string_type& default) const;

Requires: cat shall be a catalog obtained from open() and not yet closed.

Returns: A message identified by arguments set,msgid, and default, according to an implementation-defined mapping. If no such message can be found, returns defualt.

void do_close(catalog cat) const;

Requires: cat shall be a catalog obtained from open() and not yet closed.

Effects: Releases unspecified resources associated with cat.

Remarks: The limit on such resources, if any, is implementation-defined.

28.4.7.2 Class template messagesbyname

namespace std {
    template<class charT>
    class messagesbyname : public messages<charT> {
    public:
        using catalog = messages_base::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

namespace std {
    struct lconv;

         char* setlocale(int category, const char* locale);
         lconv* localeconv();
    }

#define NULL see 17.2.3
#define LC_ALL see below
#define LC_COLLATE see below
#define LC_CTYPE see below
#define LC_MONETARY see below
#define LC_NUMERIC see below
#define LC_TIME see below

1 The contents and meaning of the header <locale> are the same as the C standard library header <locale.h>.

2 Calls to the function setlocale may introduce a data race (16.5.5.9) with other calls to setlocale or with calls to the functions listed in Table 102.

See also: ISO C 7.11
Table 102 — Potential `setlocale` data races

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fprintf</code></td>
<td><code>isprint</code></td>
<td><code>iswdigit</code></td>
<td><code>localeconv</code></td>
<td><code>tolower</code></td>
</tr>
<tr>
<td><code>fscanf</code></td>
<td><code>ispunct</code></td>
<td><code>iswgraph</code></td>
<td><code>mblen</code></td>
<td><code>toupper</code></td>
</tr>
<tr>
<td><code>isalnum</code></td>
<td><code>isspace</code></td>
<td><code>iswlower</code></td>
<td><code>mbstowcs</code></td>
<td><code>towlower</code></td>
</tr>
<tr>
<td><code>isalpha</code></td>
<td><code>isupper</code></td>
<td><code>iswprint</code></td>
<td><code>mbtowc</code></td>
<td><code>towupper</code></td>
</tr>
<tr>
<td><code>isblank</code></td>
<td><code>iswalnum</code></td>
<td><code>iswpunct</code></td>
<td><code>setlocale</code></td>
<td><code>wcscoll</code></td>
</tr>
<tr>
<td><code>iscntrl</code></td>
<td><code>iswalpha</code></td>
<td><code>iswspace</code></td>
<td><code>strcoll</code></td>
<td><code>wcstod</code></td>
</tr>
<tr>
<td><code>isdigit</code></td>
<td><code>iswblank</code></td>
<td><code>iswupper</code></td>
<td><code>strerror</code></td>
<td><code>wcstombs</code></td>
</tr>
<tr>
<td><code>isgraph</code></td>
<td><code>iswcntrl</code></td>
<td><code>iswxdigit</code></td>
<td><code>strtod</code></td>
<td><code>wcsxfrm</code></td>
</tr>
<tr>
<td><code>islower</code></td>
<td><code>iswctype</code></td>
<td><code>isxdigit</code></td>
<td><code>strxfrm</code></td>
<td><code>wctomb</code></td>
</tr>
</tbody>
</table>
### 29 Input/output library

#### 29.1 General

1. This Clause describes components that C++ programs may use to perform input/output operations.

2. The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostream objects, base iostream classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 103.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>29.3</td>
<td>Forward declarations</td>
</tr>
<tr>
<td>29.4</td>
<td>Standard iostream objects</td>
</tr>
<tr>
<td>29.5</td>
<td>Iostreams base classes</td>
</tr>
<tr>
<td>29.6</td>
<td>Stream buffers</td>
</tr>
<tr>
<td>29.7</td>
<td>Formatting and manipulators</td>
</tr>
<tr>
<td>29.8</td>
<td>String streams</td>
</tr>
<tr>
<td>29.9</td>
<td>File streams</td>
</tr>
<tr>
<td>29.10</td>
<td>Synchronized output streams</td>
</tr>
<tr>
<td>29.11</td>
<td>File systems</td>
</tr>
<tr>
<td>29.12</td>
<td>C library files</td>
</tr>
</tbody>
</table>

3. 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

##### 29.2.1 Imbue limitations

1. No function described in Clause 29 except for ios_base::imbue and basic_filebuf::pubimbue causes any instance of basic_ios::imbue or basic_streambuf::imbue to be called. If any user function called from a function declared in Clause 29 or as an overriding virtual function of any class declared in Clause 29 calls imbue, the behavior is undefined.
29.2.2 Positioning type limitations

The classes of Clause 29 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos_type and traits::off_type are other types is implementation-defined. In the classes of Clause 29, a template parameter with name charT represents a member of the set of types containing char, wchar_t, and any other implementation-defined character types that satisfy the requirements for a character on which any of the iostream components can be instantiated.

29.2.3 Thread safety

Concurrent access to a stream object (29.8, 29.9), stream buffer object (29.6), or C Library stream (29.12) by multiple threads may result in a data race (6.8.2) unless otherwise specified (29.4). [Note: Data races result in undefined behavior (6.8.2). —end note]

If one thread makes a library call a that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call b such that this does not result in a data race, then a’s write synchronizes with b’s read.

29.3 Forward declarations

29.3.1 Header <iosfwd> synopsis

namespace std {
    template<class charT> struct char_traits;
    template<> struct char_traits<char>;
    template<> struct char_traits<char8_t>;
    template<> struct char_traits<char16_t>;
    template<> struct char_traits<char32_t>;
    template<> struct char_traits<wchar_t>;
    template<class T> class allocator;

    template<class charT, class traits = char_traits<charT>>
        class basic_ion;
    template<class charT, class traits = char_traits<charT>>
        class basic_streambuf;
    template<class charT, class traits = char_traits<charT>>
        class basic_istream;
    template<class charT, class traits = char_traits<charT>>
        class basic_ostream;
    template<class charT, class traits = char_traits<charT>,
             class 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 basic_filebuf;
    template<class charT, class traits = char_traits<charT>>
        class basic_ifstream;
    template<class charT, class traits = char_traits<charT>>
        class basic_ofstream;
    template<class charT, class traits = char_traits<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 istreambuf_iterator<charT> >
    class istreambuf_iterator;
template<class charT, class traits = char_traits<charT>,
    class ostreambuf_iterator<charT> >
    class ostreambuf_iterator;

using ios = basic_ios<char>;
using wios = basic_ios<wchar_t>;

using streambuf = basic_streambuf<char>;
using istream = basic_istream<char>;
using ostream = basic_ostream<char>;
using iostream = basic_iostream<char>;

using stringbuf = basic_stringbuf<char>;
using iostreamstream = basic_iostreamstream<char>;
using ostrstream = basic_ostringstream<char>;
using stringstream = basic_stringstream<char>;

using filebuf = basic_filebuf<char>;
using ifstream = basic_ifstream<char>;
using ofstream = basic_ofstream<char>;
using fstream = basic_fstream<char>;

using syncbuf = basic_syncbuf<char>;
using osyncstream = basic_osyncstream<char>;

using ustringbuf = basic_streambuf<wchar_t>;
using uistream = basic_istream<wchar_t>;
using uostream = basic_ostream<wchar_t>;
using uiostream = basic_iostream<wchar_t>;

using ustringstream = basic_stringstream<wchar_t>;
using uistringstream = basic_istringstream<wchar_t>;
using uostringstream = basic_ostringstream<wchar_t>;
using wstringstream = basic_stringstream<wchar_t>;
using filebuf = basic_filebuf<wchar_t>;
using ifstream = basic_ifstream<wchar_t>;
using ofstream = basic_ofstream<wchar_t>;
using fstream = basic_fstream<wchar_t>;

using syncbuf = basic_syncbuf<wchar_t>;
using osyncstream = basic_osyncstream<wchar_t>;

using ustringbuf = basic_streambuf<char8_t>;
using uostream = basic_ostream<char8_t>;
using uostreampos = fpos<char_traits<char8_t>::state_type>;
using ustringstream = basic_ustringstream<char16_t>;
using uostreampos = fpos<char_traits<char16_t>::state_type>;
using ustringstream = basic_ustringstream<char32_t>;
using uostreampos = fpos<char_traits<char32_t>::state_type>;

1 Default template arguments are described as appearing both in <iosfwd> and in the synopsis of other headers but it is well-formed to include both <iosfwd> and one or more of the other headers.

287) It is the implementation’s responsibility to implement headers so that including <iosfwd> and other headers does not violate the rules about multiple occurrences of default arguments.

§ 29.3.1 1313
29.3.2 Overview

1 The class template specialization basic_ios<charT, traits> serves as a virtual base class for the class templates basic_istream, basic_ostream, and class templates derived from them. basic_iostream is a class template derived from both basic_istream<charT, traits> and basic_ostream<charT, traits>.

2 The class template specialization basic_streambuf<charT, traits> serves as a base class for class templates basic_stringbuf, basic_filebuf, and basic_syncbuf.

3 The class template specialization basic_istream<charT, traits> serves as a base class for class templates basic_iostream and basic_ifstream.

4 The class template specialization basic_ostream<charT, traits> serves as a base class for class templates basic_ostream and basic_osyncstream.

5 The class template specialization basic_iostream<charT, traits> serves as a base class for class templates basic_iostream and basic_fstream.

6 [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]

7 Other typedef-names define instances of class templates specialized for char or wchar_t types.

8 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]

9 [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 <iostream> // see 29.7.1
#include <ostream> // see 29.7.2

namespace std {
    extern istream cin;
    extern ostream cout;
    extern ostream cerr;
    extern ostream clog;

    extern wistream wcin;
    extern wostream wcout;
    extern wostream wcerr;
    extern wostream wclog;
}
```

29.4.2 Overview

1 In this Clause, the type name FILE refers to the type FILE declared in <cstdio> (29.12.1).

2 The header <iostream> declares objects that associate objects with the standard C streams provided for by the functions declared in <cstdio> (29.12), and includes all the headers necessary to use these objects.

3 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 (6.8.3.1) 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.

4 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.

---

288) If it is possible for them to do so, implementations should initialize the objects earlier than required.
289) Constructors and destructors for static objects 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 shall not result in a data race (6.8.2). 

[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;
```
1. The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (29.12.1).

2. 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;
```
3. The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (29.12.1).

```cpp
ostream cerr;
```
4. The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

5. 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;
```
6. 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;
```
1. The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (29.12.1).

2. 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;
```
3. The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (29.12.1).

```cpp
wostream wcerr;
```
4. The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

5. 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;
```
6. The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (29.12.1).

29.5 Iostreams base classes

29.5.1 Header `<ios>` synopsis

```cpp
#include <iosfwd> // see 29.3.1

namespace std {
    using streamoff = implementation-defined;
    using streamsize = implementation-defined;
} // namespace std
```

§ 29.5.1
template<class stateT> class fpos;

class ios_base;

template<class charT, class traits = char_traits<charT>>
class basic_ios;

// 29.5.6, manipulators
ios_base& boolalpha (ios_base& str);
ios_base& noboolalpha(ios_base& str);

ios_base& showbase (ios_base& str);
ios_base& noshowbase (ios_base& str);

ios_base& showpoint (ios_base& str);
ios_base& noshowpoint(ios_base& str);

ios_base& showpos (ios_base& str);
ios_base& noshowpos (ios_base& str);

ios_base& skipws (ios_base& str);
ios_base& noskipws (ios_base& str);

ios_base& uppercase (ios_base& str);
ios_base& nouppercase(ios_base& str);

ios_base& unitbuf (ios_base& str);
ios_base& nounitbuf (ios_base& str);

// 29.5.6.2, adjustfield
ios_base& internal (ios_base& str);
ios_base& left (ios_base& str);
ios_base& right (ios_base& str);

// 29.5.6.3, basefield
ios_base& dec (ios_base& str);
ios_base& hex (ios_base& str);
ios_base& oct (ios_base& str);

// 29.5.6.4, floatfield
ios_base& fixed (ios_base& str);
ios_base& scientific (ios_base& str);
ios_base& hexfloat (ios_base& str);
ios_base& defaultfloat(ios_base& str);

// 29.5.7, error reporting
enum class io_errc {
  stream = 1
};

template<> struct is_error_code_enum<io_errc> : public true_type { };  
error_code make_error_code(io_errc e) noexcept;
error_condition make_error_condition(io_errc e) noexcept;
const error_category& iostream_category() noexcept;
}

29.5.2 Types [stream.types]

using streamoff = implementation-defined;

1 The type streamoff is a synonym for one of the signed basic integral types of sufficient size to represent the maximum possible file size for the operating system.290

290) Typically long long.
using streamsize = implementation-defined;

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.\(^{291}\)

### 29.5.3 Class `ios_base`

```cpp
namespace std {
  class ios_base {
  public:
    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);
  }
}
```

\(^{291}\)`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

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 it shall 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]

    explicit failure(const string& msg, const error_code& ec = io_errc::stream);
3 Effects: Constructs an object of class failure by constructing the base class with msg and ec.

    explicit failure(const char* msg, const error_code& ec = io_errc::stream);
4 Effects: Constructs an object of class failure by constructing the base class with msg and ec.

29.5.3.1.2 Type ios_base::fmtflags

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 104.

<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 105.
Table 105 — 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 [ios.iostate]

using iostate = T2;

1 The type iostate is a bitmask type (16.4.2.4) that contains the elements indicated in Table 106.

Table 106 — 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 [ios.openmode]

using openmode = T3;

1 The type openmode is a bitmask type (16.4.2.4). It contains the elements indicated in Table 107.

Table 107 — 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>ate</td>
<td>open and seek to end immediately after opening</td>
</tr>
<tr>
<td>binary</td>
<td>perform input and output in binary mode (as opposed to text mode)</td>
</tr>
<tr>
<td>in</td>
<td>open for input</td>
</tr>
<tr>
<td>out</td>
<td>open for output</td>
</tr>
<tr>
<td>trunc</td>
<td>truncate an existing stream when opening</td>
</tr>
</tbody>
</table>

29.5.3.1.5 Type ios_base::seekdir [ios.seekdir]

using seekdir = T4;

1 The type seekdir is an enumerated type (16.4.2.3) that contains the elements indicated in Table 108.

Table 108 — 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

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
    };
  }
}

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).

For the sake of exposition, the maintained data is presented here as:

— static int init_cnt, counts the number of constructor and destructor calls for class Init, initialized to zero.

Init();

Effects: Constructs an object of class Init. Constructs and initializes the objects cin, cout, cerr, clog, wcin, wcout, wcerr, and wclog if they have not already been constructed and initialized.

~Init();

Effects: Destroys an object of class Init. 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

fmtflags flags() const;

Returns: The format control information for both input and output.

fmtflags flags(fmtflags fmtfl);

Ensures: fmtfl == flags().

Returns: The previous value of flags().

fmtflags setf(fmtflags fmtfl);

Effects: Sets fmtfl in flags().

Returns: The previous value of flags().

fmtflags setf(fmtflags fmtfl, fmtflags mask);

Effects: Clears mask in flags(), sets fmtfl & mask in flags().

Returns: The previous value of flags().

void unsetf(fmtflags mask);

Effects: Clears mask in flags().

streamsize precision() const;

Returns: The precision to generate on certain output conversions.

streamsize precision(streamsize prec);

Ensures: prec == precision().

Returns: The previous value of precision().

streamsize width() const;

Returns: The minimum field width (number of characters) to generate on certain output conversions.
streamsize width(streamsize wide);

Ensures: wide == width().

Returns: The previous value of width().

29.5.3.3 Functions

locale imbue(const locale& loc);

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.

Returns: The previous value of getloc().

Ensures: loc == getloc().

locale getloc() const;

Returns: The previous value of getloc().

Ensures: loc == getloc().

29.5.3.4 Static members

bool sync_with_stdio(bool sync = true);

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.

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.

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()->sbump();

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.

29.5.3.5 Storage functions

static int xalloc();

Returns: index ++.

Remarks: Concurrent access to this function by multiple threads shall not result in a data race (6.8.2).

long& iword(int idx);

Requires: idx is a value obtained by a call to xalloc.

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

292) 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 \texttt{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 \texttt{copyfmt}, calling \texttt{ivord} with the same index yields another reference to the same value. If the function fails and *\texttt{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).

\textit{Returns:} On success \texttt{iarray[idx]}. On failure, a valid \texttt{long} initialized to 0.

\texttt{void*\& pword(int idx);}  
\textit{Requires:} \texttt{idx} is a value obtained by a call to \texttt{xalloc}.

\textit{Effects:} If \texttt{parray} is a null pointer, allocates an array of pointers to \texttt{void} of unspecified size and stores a pointer to its first element in \texttt{parray}. The function then extends the array pointed at by \texttt{parray} as necessary to include the element \texttt{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 \texttt{copyfmt}, calling \texttt{pword} with the same index yields another reference to the same value. If the function fails and *\texttt{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).

\textit{Returns:} On success \texttt{parray[idx]}. On failure a valid \texttt{void*\&} initialized to 0.

\textit{Remarks:} After a subsequent call to \texttt{pword(int)} for the same object, the earlier return value may no longer be valid.

\subsection{29.5.3.6 Callbacks} [ios.base.callback]

\texttt{void register_callback(event\_callback fn, int idx);}  
\textit{Effects:} Registers the pair (\texttt{fn, idx}) such that during calls to \texttt{imbue()} (29.5.3.3), \texttt{copyfmt()}, or \texttt{\~ios\_base()} (29.5.3.7), the function \texttt{fn} is called with argument \texttt{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.

\textit{Requires:} The function \texttt{fn} shall not throw exceptions.

\textit{Remarks:} Identical pairs are not merged. A function registered twice will be called twice.

\subsection{29.5.3.7 Constructors and destructor} [ios.base.cons]

\texttt{ios\_base();}  
\textit{Effects:} Each \texttt{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.

\texttt{\~ios\_base();}  
\textit{Effects:} Destroys an object of class \texttt{ios\_base}. Calls each registered callback pair (\texttt{fn, idx}) (29.5.3.6) as \texttt{(*fn)(erase\_event, \*\texttt{this}, idx)} at such time that any \texttt{ios\_base} member function called from within \texttt{fn} has well-defined results.

\subsection{29.5.4 Class template fpos} [fpos]

\begin{verbatim}
namespace std {
    template<class stateT> class fpos {
    public:
        // 29.5.4.1, members
        stateT state() const;
        void state(stateT);  
    private:
        stateT st; // exposition only
    }
}
\end{verbatim}

293) An implementation is free to implement both the integer array pointed at by \texttt{iarray} and the pointer array pointed at by \texttt{parray} as sparse data structures, possibly with a one-element cache for each.

294) For example, because it cannot allocate space.

295) For example, because it cannot allocate space.
29.5.4.1 Members

```cpp
void state(stateT s);
```

1. **Effects:** Assigns `s` to `st`.

```cpp
stateT state() const;
```

2. **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 25), `Cpp17CopyConstructible` (Table 27), `Cpp17CopyAssignable` (Table 29), and `Cpp17Destructible` (Table 30) requirements. If `is_trivially_copy_constructible_v<stateT>` is true, then `fpos<stateT>` has a trivial copy constructor. If `is_trivially_copyAssignable<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` satisfy the `Cpp17DefaultConstructible`, `Cpp17CopyConstructible`, `Cpp17CopyAssignable`, `Cpp17Destructible`, and `Cpp17EqualityComparable` (Table 23) requirements. In addition, the expressions shown in Table 109 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 `ql` refer to modifiable lvalues of type `P`,
- **(1.4)** — `o` refers to type `streamoff`, and
- **(1.5)** — `o` refers to a value of type `streamoff` or `const streamoff`.

![Table 109 — Position type requirements](image)

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.

§ 29.5.4.2
29.5.5 Class template basic_ios

29.5.5.1 Overview

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

```cpp
explicit basic_ios(basic_streambuf<charT, traits>* sb);
```

**Effects:** Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```
basic_ios();
```

**Effects:** Constructs an object of class `basic_ios` (29.5.3.7) leaving 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.

```cpp
~basic_ios();
```

**Remarks:** The destructor does not destroy `rdbuf()`.

```cpp
void init(basic_streambuf<charT, traits>* sb);
```

**Ensures:** The postconditions of this function are indicated in Table 110.

<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

```cpp
basic_ostream<charT, traits>* tie() const;
```

**Returns:** An output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.

```cpp
basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);
```

**Requires:** If `tiestr` is not null, `tiestr` shall not be reachable by traversing the linked list of tied stream objects starting from `tiestr->tie()`.  

**Ensures:** `tiestr == tie()`.

**Returns:** The previous value of `tie()`.

```cpp
basic_streambuf<charT, traits>* rdbuf() const;
```

**Returns:** A pointer to the `streambuf` associated with the stream.

```cpp
basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);
```

**Ensures:** `sb == rdbuf()`.

**Effects:** Calls `clear()`.

**Returns:** The previous value of `rdbuf()`.

```cpp
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).
10 Returns: The prior value of `ios_base::imbue()`.

```cpp
char narrow(char_type c, char dfault) const;
```

Returns: use_facet<ctype<char_type>>(getloc()).narrow(c, dfault)

```cpp
char_type widen(char c) const;
```

Returns: use_facet<ctype<char_type>>(getloc()).widen(c)

```cpp
char_type fill() const;
```

Returns: The character used to pad (fill) an output conversion to the specified field width.

```cpp
char_type fill(char_type fillch);
```

Ensures: traits::eq(fillch, fill()).

Returns: The previous value of fill().

```cpp
basic_ios& copyfmt(const basic_ios& rhs);
```

Effects: If (this == addressof(rhs)) does nothing. Otherwise assigns to the member objects of *this the corresponding member objects of rhs as follows:

(16.1) — calls each registered callback pair (fn, idx) as (*fn)(erase_event, *this, idx);

(16.2) — then, assigns to the member objects of *this the corresponding member objects of rhs, except that

(16.2.1) — rdstate(), rdbuf(), and exceptions() are left unchanged;

(16.2.2) — the contents of arrays pointed at by pword and iword are copied, not the pointers themselves;

(16.2.3) — if any newly stored pointer values in *this point at objects stored outside the object rhs and those objects are destroyed when rhs is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects;

(16.3) — then, calls each callback pair that was copied from rhs as (*fn)(copyfmt_event, *this, idx);

(16.4) — then, calls exceptions(rhs.exceptions()).

[Note: The second pass through the callback pairs permits a copied pword value to be zeroed, or to have its referent deep copied or reference counted, or to have other special action taken. — end note]

Ensures: The postconditions of this function are indicated in Table 111.

<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.

```cpp
void move(basic_ios& rhs);
```

296) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is nonzero.
void move(basic_ios&& rhs);

  Ensures: *this shall have the state that rhs had before the function call, except that rdbuf() shall return 0. rhs shall be in a valid but unspecified state, except that rhs.rdbuf() shall return the same value as it returned before the function call, and rhs.tie() shall return 0.

void swap(basic_ios& rhs) noexcept;

  Effects: The states of *this and rhs shall be exchanged, except that rdbuf() shall return the same value as it returned before the function call, and rhs.rdbuf() shall return the same value as it returned before the function call.

void set_rdbuf(basic_streambuf<charT, traits>* sb);

  Requires: sb != nullptr.

  Effects: Associates the basic_streambuf object pointed to by sb with this stream without calling clear().

  Ensures: rdbuf() == sb.

  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);

  Ensures: 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 basic_ios::failure (29.5.3.1.1), constructed with implementation-defined argument values.

void setstate(iostate state);

  Effects: Calls clear(rdstate() | state) (which may throw basic_ios::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);

  Ensures: except == exceptions().

297) Checking badbit also for fail() is historical practice.
Effects: Calls clear(rdstate()).

29.5.6  

ios_base manipulators

29.5.6.1  

fmtflags manipulators

Each function specified in this subclause is a designated addressable function (16.5.4.2.1).

ios_base& boolalpha(ios_base& str);

Effects: Calls str.setf(ios_base::boolalpha).

Returns: str.

ios_base& noboolalpha(ios_base& str);

Effects: Calls str.unsetf(ios_base::boolalpha).

Returns: str.

ios_base& showbase(ios_base& str);

Effects: Calls str.setf(ios_base::showbase).

Returns: str.

ios_base& noshowbase(ios_base& str);

Effects: Calls str.unsetf(ios_base::showbase).

Returns: str.

ios_base& showpoint(ios_base& str);

Effects: Calls str.setf(ios_base::showpoint).

Returns: str.

ios_base& noshowpoint(ios_base& str);

Effects: Calls str.unsetf(ios_base::showpoint).

Returns: str.

ios_base& showpos(ios_base& str);

Effects: Calls str.setf(ios_base::showpos).

Returns: str.

ios_base& noshowpos(ios_base& str);

Effects: Calls str.unsetf(ios_base::showpos).

Returns: str.

ios_base& skipws(ios_base& str);

Effects: Calls str.setf(ios_base::skipws).

Returns: str.

ios_base& noskipws(ios_base& str);

Effects: Calls str.unsetf(ios_base::skipws).

Returns: str.

ios_base& uppercase(ios_base& str);

Effects: Calls str.setf(ios_base::uppercase).

Returns: str.

ios_base& nouppercase(ios_base& str);

Effects: Calls str.unsetf(ios_base::uppercase).

Returns: str.
ios_base& unitbuf(ios_base& str);
   Effects: Calls str.setf(ios_base::unitbuf).
   Returns: str.

ios_base& nounitbuf(ios_base& str);
   Effects: Calls str.unsetf(ios_base::unitbuf).
   Returns: 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);
   Effects: Calls str.setf(ios_base::internal, ios_base::adjustfield).
   Returns: str.

ios_base& left(ios_base& str);
   Effects: Calls str.setf(ios_base::left, ios_base::adjustfield).
   Returns: str.

ios_base& right(ios_base& str);
   Effects: Calls str.setf(ios_base::right, ios_base::adjustfield).
   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);
   Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
   Returns: str

ios_base& hex(ios_base& str);
   Effects: Calls str.setf(ios_base::hex, ios_base::basefield).
   Returns: str.

ios_base& oct(ios_base& str);
   Effects: Calls str.setf(ios_base::oct, ios_base::basefield).
   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);
   Effects: Calls str.setf(ios_base::fixed, ios_base::floatfield).
   Returns: str.

ios_base& scientific(ios_base& str);
   Effects: Calls str.setf(ios_base::scientific, ios_base::floatfield).
   Returns: str.

ios_base& hexfloat(ios_base& str);
   Effects: Calls str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield).
   Returns: str.

298) The function signature dec(ios_base&) can be called by the function signature basic_ostream& stream::operator<<(ios_base& (*)(ios_base&)) to permit expressions of the form cout << dec to change the format flags stored in cout.
[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]

```cpp
ios_base& defaultfloat(ios_base& str);
```

Effects: Calls `str.unsetf(ios_base::floatfield)`.

Returns: `str`.

### 29.5.7 Error reporting

```cpp
error_code make_error_code(io_errc e) noexcept;
```

Returns: `error_code(static_cast<int>(e), iostream_category())`.

```cpp
error_condition make_error_condition(io_errc e) noexcept;
```

Returns: `error_condition(static_cast<int>(e), iostream_category())`.

```cpp
const error_category& iostream_category() noexcept;
```

Returns: A reference to an object of a type derived from class `error_category`.

The object’s `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object’s `name` virtual function 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>;
}
```

The header `<streambuf>` defines types that control input from and output to `character` sequences.

#### 29.6.2 Stream buffer requirements

Stream buffers can impose various constraints on the sequences they control. Some constraints are:

1. The controlled input sequence can be not readable.
2. The controlled output sequence can be not writable.
3. The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.
4. The controlled sequences can support operations directly to or from associated sequences.
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.

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:

1. the `beginning pointer`, or lowest element address in the array (called `xbeg` here);
2. the `next pointer`, or next element address that is a current candidate for reading or writing (called `xnext` here);
3. the `end pointer`, or first element address beyond the end of the array (called `xend` here).

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 \texttt{xnext} is not a null pointer, then \texttt{xbeg} and \texttt{xend} shall also be non-null pointers into the same \texttt{charT} array, as described above; otherwise, \texttt{xbeg} and \texttt{xend} shall also be null.

(3.2) If \texttt{xnext} is not a null pointer and \texttt{xnext < xend} for an output sequence, then a \textit{write position} is available. In this case, \texttt{*xnext} shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

(3.3) If \texttt{xnext} is not a null pointer and \texttt{xbeg < xnext} for an input sequence, then a \textit{putback position} is available. In this case, \texttt{xnext[\texttt{-}1]} shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.

(3.4) 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).

29.6.3 Class template basic_streambuf

namespace std {

    template<class charT, class traits = char_traits<charT>>
    class basic_streambuf {
        public:
            using char_type = charT;
            using int_type = typename traits::int_type;
            using pos_type = typename traits::pos_type;
            using off_type = typename traits::off_type;
            using traits_type = traits;
            virtual ~basic_streambuf();

            // 29.6.3.2.1, 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);

    };
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

```
basic_streambuf();
```

**Effects:** Constructs an object of class `basic_streambuf<charT, traits>` and initializes:

1. (1.1) — all its 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.

---

299) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.
basic_streambuf(const basic_streambuf& rhs);

Effects: Constructs a copy of rhs.

Ensures:

1. eback() == rhs.eback()
2. gptr() == rhs.gptr()
3. egptr() == rhs.egptr()
4. pbase() == rhs.pbase()
5. pptr() == rhs.pptr()
6. eptr() == rhs.eptr()
7. 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);

Ensures: loc == getloc().

Effects: Calls imbue(loc).

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., 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();

4 Returns: 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();

5 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);

6 Returns: xsgetn(s, n).

29.6.3.2.4 Putback

int_type sputbackc(char_type c);

1 Returns: If the input sequence putback position is not available, or if traits::eq(c, gptr()[-1]) is false, returns pbackfail(traits::to_int_type(c)). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

int_type sungetc();

2 Returns: If the input sequence putback position is not available, returns pbackfail(). Otherwise, decrements the next pointer for the input sequence and returns traits::to_int_type(*gptr()).

29.6.3.2.5 Put area

int_type sputc(char_type c);

1 Returns: If the output sequence write position is not available, returns overflow(traits::to_int_type(c)). Otherwise, stores c at the next pointer for the output sequence, increments the pointer, and returns traits::to_int_type(c).

streamsize sputn(const char_type* s, streamsize n);

2 Returns: xsputn(s, n).

29.6.3.3 Protected member functions

29.6.3.3.1 Assignment

basic_streambuf& operator=(const basic_streambuf& rhs);

1 Effects: Assigns the data members of rhs to *this.

2 Ensures:

(2.1) — eback() == rhs.eback()
(2.2) — gptr() == rhs.gptr()
(2.3) — egptr() == rhs.egptr()
(2.4) — pbase() == rhs.pbase()
(2.5) — pptr() == rhs.pptr()
(2.6) — epptr() == rhs.epptr()
(2.7) — getloc() == rhs.getloc()

3 Returns: *this.

void swap(basic_streambuf& rhs);

4 Effects: Swaps the data members of rhs and *this.

29.6.3.3.2 Get area access

char_type* eback() const;

1 Returns: The beginning pointer for the input sequence.
```cpp
char_type* gptr() const;
Returns: The next pointer for the input sequence.

char_type* egptr() const;
Returns: The end pointer for the input sequence.

void gbump(int n);
Effects: Adds n to the next pointer for the input sequence.

void setg(char_type* gbeg, char_type* gnext, char_type* gend);
Ensures: gbeg == eback(), gnext == gptr(), and gend == egptr().

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);
Ensures: pbeg == pbase(), pbeg == pptr(), and pend == epptr().

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 \texttt{pos\_type(off\_type(-1))}.

```c
int sync();
```

\textit{Effects:} Synchronizes the controlled sequences with the arrays. That is, if \texttt{pbase()} is non-null the characters between \texttt{pbase()} and \texttt{pptr()} are written to the controlled sequence. The pointers may then be reset as appropriate.

\textit{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 \texttt{[streambuf.virt.get]}

```c
streamsize showmanyc();
```

\textit{Returns:} An estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to \texttt{underflow()} will not return \texttt{traits::eof()} until at least that number of characters have been extracted from the stream. If \texttt{showmanyc()} returns -1, then calls to \texttt{underflow()} or \texttt{uflow()} will fail. 301

Default behavior: Returns zero.

Remarks: Uses \texttt{traits::eof()}. 302

```c
streamsize xsgetn(char\_type* s, streamsize n);
```

\textit{Effects:} Assigns up to \texttt{n} characters to successive elements of the array whose first element is designated by \texttt{s}. The characters assigned are read from the input sequence as if by repeated calls to \texttt{sbumpc()}. Assigning stops when either \texttt{n} characters have been assigned or a call to \texttt{sbumpc()} would return \texttt{traits::eof()}. 302

Returns: The number of characters assigned. 302

Remarks: Uses \texttt{traits::eof()}. 302

```c
int\_type underflow();
```

\textit{Remarks:} The public members of \texttt{basic\_streambuf} call this virtual function only if \texttt{gptr()} is null or \texttt{gptr()} \texttt{\geq egptr()}. 302

Returns: \texttt{traits::to\_int\_type(c)}, where \texttt{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 \texttt{traits::eof()} to indicate failure.

The pending sequence of characters is defined as the concatenation of

\begin{itemize}
  \item the empty sequence if \texttt{gptr()} is null, otherwise the characters in \texttt{[gptr(), egptr())}, followed by
  \item some (possibly empty) sequence of characters read from the input sequence.
\end{itemize}

The result character is the first character of the pending sequence if it is non-empty, otherwise the next character that would be read from the input sequence.

The backup sequence is the empty sequence if \texttt{eback()} is null, otherwise the characters in \texttt{[eback(), gptr())}. 302

\textit{Effects:} The function sets up the \texttt{gptr()} and \texttt{egptr()} such that if the pending sequence is non-empty, then \texttt{egptr()} is non-null and the characters in \texttt{[gptr(), egptr())} are the characters in the pending sequence, otherwise either \texttt{gptr()} is null or \texttt{gptr() == egptr()}. 302

If \texttt{eback()} and \texttt{gptr()} are non-null then the function is not constrained as to their contents, but the “usual backup condition” is that either

\begin{itemize}
  \item the backup sequence contains at least \texttt{gptr()} - \texttt{eback()} characters, in which case the characters in \texttt{[eback(), gptr())} agree with the last \texttt{gptr()} - \texttt{eback()} characters of the backup sequence, or
\end{itemize}

300) The morphemes of \texttt{showmanyc} are “es-how-many-see”, not “show-manic”.

301) \texttt{underflow} or \texttt{uflow} might fail by throwing an exception prematurely. The intention is not only that the calls will not return \texttt{eof()} but that they will return “immediately”.

302) Classes derived from \texttt{basic\_streambuf} can provide more efficient ways to implement \texttt{xsgetn()} and \texttt{xsputn()} by overriding these definitions from the base class.
the characters in \([\text{gptr}() - n, \text{gptr}()]\) agree with the backup sequence (where \(n\) is the length of the backup sequence).

**Default behavior:** Returns `traits::eof()`.

```
int_type uflow();
```

**Requires:** The constraints are the same as for `underflow()`, except that the result character shall be transferred from the pending sequence to the backup sequence, and the pending sequence shall not be empty before the transfer.

**Default behavior:** Calls `underflow()`. If `underflow()` returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns the value of `traits::to_int_type(*\text{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

**[streambuf.virt.pback]**

```
int_type pbackfail(int_type c = traits::eof());
```

**Remarks:** The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `traits::eq(traits::to_char_type(c), gptr()[\-1])` returns `false`. Other calls shall also satisfy that constraint.

The *pending sequence* is defined as for `underflow()`, with the modifications that

1. If `traits::eq_int_type(c, traits::eof())` returns `true`, then the input sequence is backed up one character before the pending sequence is determined.
2. If `traits::eq_int_type(c, traits::eof())` returns `false`, then \(c\) is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

**Ensures:** 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

**[streambuf.virt.put]**

```
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.

**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.

**Requires:** Every overriding definition of this virtual function shall obey the following constraints:
(5.1) The effect of consuming a character on the associated output sequence is specified.\[303\]

(5.2) Let \( r \) be the number of characters in the pending sequence not consumed. If \( r \) is nonzero then \( \text{pbase()} \) and \( \text{pptr()} \) shall be set so that: \( \text{pptr()} - \text{pbase()} = r \) and the \( r \) characters starting at \( \text{pbase()} \) are the associated output stream. In case \( r \) is zero (all characters of the pending sequence have been consumed) then either \( \text{pbase()} \) is set to \( \text{nullptr} \), or \( \text{pbase()} \) and \( \text{pptr()} \) are both set to the same non-null value.

(5.3) The function may fail if either appending some character to the associated output stream fails or if it is unable to establish \( \text{pbase()} \) and \( \text{pptr()} \) according to the above rules.

Returns: \( \text{traits::eof()} \) or throws an exception if the function fails.
Otherwise, returns some value other than \( \text{traits::eof()} \) to indicate success.\[304\]

Default behavior: Returns \( \text{traits::eof()} \).

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_ostream;

    using ostream = basic_ostream<char>;
    using wostream = basic_ostream<wchar_t>;

    template<class charT, class traits, class T>
    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);
}
```

303] 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.

304] Typically, `overflow` returns \( c \) to indicate success, except when \( \text{traits::eq_int_type}(c, \text{traits::eof()}) \) returns true, in which case it returns \( \text{traits::not_eof}(c) \).
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

namespace std {

    // types T1, T2, ... are unspecified implementation types
    T1 resetiosflags(ios_base::fmtflags mask);
    T2 setiosflags (ios_base::fmtflags mask);
    T3 setbase(int base);
    template<class charT> T4 setfill(charT c);
    T5 setprecision(int n);
    T6 setw(int n);
    template<class moneyT> T7 get_money(moneyT& mon, bool intl = false);
    template<class moneyT> T8 put_money(const moneyT& mon, bool Intl = false);
    template<class charT> T9 get_time(struct tm* tmb, const charT* fmt);
    template<class charT> T10 put_time(const struct tm* tmb, const charT* fmt);

    template<class charT>
    T11 quoted(const charT* s, charT delim = charT('"'), charT escape = charT(\\));

    template<class charT, class traits, class Allocator>
    T12 quoted(const basic_string<charT, traits, Allocator>& s,
                charT delim = charT('"'), charT escape = charT(\\));

    template<class charT, class traits>
    T13 quoted(basic_string_view<charT, traits> s,
                charT delim = charT('"'), charT escape = charT(\\));

    template<class charT, class traits, class T>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&& os, const T& x);
}

29.7.4 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

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_istream<charT, traits>&& (*pf)(basic_ios<charT, traits>&&));

§ 29.7.4.1 1340
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);
basic_istream<charT, traits>& operator>>(unsigned long& n);
basic_istream<charT, traits>& operator>>(long long& n);
basic_istream<charT, traits>& operator>>(unsigned long long& n);
basic_istream<charT, traits>& operator>>(float& f);
basic_istream<charT, traits>& operator>>(double& f);
basic_istream<charT, traits>& operator>>(long double& f);

basic_istream<charT, traits>& operator>>(void*& p);
basic_istream<charT, traits>& operator>>(basic_streambuf<char_type, traits>* sb);

// 29.7.4.3, unformatted input
streamsize gcount() const;
int_type get();

basic_istream<charT, traits>& get(char_type& c);
basic_istream<charT, traits>& get(char_type* s, streamsize n);
basic_istream<charT, traits>& get(char_type* s, streamsize n, char_type delim);

basic_istream<charT, traits>& getline(char_type* s, streamsize n);
basic_istream<charT, traits>& getline(char_type* s, streamsize n, char_type delim);

basic_istream<charT, traits>& ignore(streamsize n = 1, int_type delim = traits::eof());
int_type peek();

basic_istream<charT, traits>& read (char_type* s, streamsize n);
streamsize readsome(char_type* s, streamsize n);

basic_istream<charT, traits>& putback(char_type c);
basic_istream<charT, traits>& unget();

int sync();
pow_type tellg();

basic_istream<charT, traits>& seekg(pow_type);

protected:
// 29.7.4.1.1, copy/move constructor
basic_istream(const basic_istream& rhs) = delete;
basic_istream(basic_istream&& rhs);

// 29.7.4.1.2, assign and swap
basic_istream& operator=(const basic_istream& rhs) = delete;
basic_istream& operator=(basic_istream&& rhs);
void swap(basic_istream& rhs);

// 29.7.4.2.3, character extraction templates
template<class charT, class traits>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, charT&);
template<class traits>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, unsigned char&);
template<class traits>
  basic_istream<char, traits>& operator>>(basic_istream<char, traits>&, signed char&);

§ 29.7.4.1
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 error state. If badbit is on 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
template<class charT, class traits, size_t N>
    basic_istream<CharT, traits>& operator>>(basic_istream<CharT, traits>&, charT(&)[N]);
```  
1 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.

The effects of calling these functions are as follows:

1. **explicit basic_istream(basic_streambuf<CharT, traits>* sb);**
   - **Effects:** Constructs an object of class basic_istream, initializing the base class subobject with basic_ios::init(sb) (29.5.5.2).
   - **Ensures:** gcoun() == 0.

2. **basic_istream(basic_istream&& rhs);**
   - **Effects:** Move constructs from the rvalue rhs. This is accomplished by default constructing the base class, copying the gcoun() from rhs, calling basic_ios<CharT, traits>::move(rhs) to initialize the base class, and setting the gcoun() for rhs to 0.

3. **virtual ~basic_istream();**
   - **Effects:** Destroys an object of class basic_istream.
   - **Remarks:** Does not perform any operations of rdbuf().

### 29.7.4.1.2 Assignment and swap

```cpp
basic_istream& operator=(basic_istream&& rhs);
```  
1 The effects of calling these functions are as follows:

1. **basic_istream& operator=(basic_istream&& rhs);**
   - **Effects:** As if by swap(rhs).
   - **Returns:** *this.

### 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 {
    using traits_type = traits;
    bool ok_; // exposition only
    public:
    explicit sentry(basic_istream<CharT, traits>& is, bool noskipws = false);
    ~sentry();
    explicit operator bool() const { return ok_; }
    sentry(const sentry&) = delete;
```
sentry& operator=(const sentry&) = delete;
};

The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```cpp
explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);
```

**Effects:** If `is.good()` is `false`, calls `is.setstate(failbit)`. Otherwise, prepares for formatted or unformatted input. First, if `is.tie()` is not a null pointer, the function calls `is.tie()->flush()` to synchronize the output sequence with any associated external C stream. Except that this call can be suppressed if the put area of `is.tie()` is empty. Further an implementation is allowed to defer the call to `flush` until a call of `is.rdbuf()->underflow()` occurs. If no such call occurs before the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.\(^3\)\(^0\)\(^5\) 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()->sgetc()` or `is.rdbuf()->sbumpc()` 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_` is `false` otherwise, `ok_` is `false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure`)\(^3\)\(^0\)\(^6\)

```cpp
~sentry();
```

**Effects:** None.

```cpp
explicit operator bool() const;
```

**Effects:** Returns `ok_`.

### 29.7.4.2 Formatted input functions

#### 29.7.4.2.1 Common requirements

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::badbit` is turned on\(^3\)\(^0\)\(^7\) 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

```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);
```

\(^3\)\(^0\)\(^5\) 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.

\(^3\)\(^0\)\(^6\) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

\(^3\)\(^0\)\(^7\) This is done without causing an `ios::failure` to be thrown.
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:

```
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);
```

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.

operator>>(short& val);

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iosstate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (lval < numeric_limits<short>::min()) {
  err |= ios_base::failbit;
  val = numeric_limits<short>::min();
} else if (numeric_limits<short>::max() < lval) {
  err |= ios_base::failbit;
  val = numeric_limits<short>::max();
} else
  val = static_cast<short>(lval);
setstate(err);
```

operator>>(int& val);

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iosstate err = ios_base::goodbit;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (lval < numeric_limits<int>::min()) {
  err |= ios_base::failbit;
  val = numeric_limits<int>::min();
} else if (numeric_limits<int>::max() < lval) {
  err |= ios_base::failbit;
  val = numeric_limits<int>::max();
} else
  val = static_cast<int>(lval);
setstate(err);
```

### 29.7.4.2.3 basic_istream::operator>>(basic_istream<>&) [istream.extractors]

```
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&));
```

**Effects:** None. This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

**Returns:** pf(*this).308

308) See, for example, the function signature ws(basic_istream&) (29.7.4.4).
basic_istream<char, traits>&
operator>>(basic_istream<char, traits>& (*pf)(basic_istream<char, traits>&&));

Effects: Calls pf(*this). This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

Returns: *this.

basic_istream<char, traits>& operator>>(ios_base& (*pf)(ios_base&));

Effects: Calls pf(*this). This extractor does not behave as a formatted input function (as described in 29.7.4.2.1).

Returns: *this.

309] 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]);

template<class traits, size_t N>
basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, unsigned char (&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, operator>> extracts characters and stores them into s. If width() is greater than zero, \( n = \min(\text{size}_t(\text{width}()), N) \). Otherwise \( n = N \). \( n \) 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. letting \( ct \) be use_facet<typename<charT>>(in.getloc()), \( ct.\text{is}(ct.\text{space}, c) \) is true.

operator>> then stores a null byte (charT()) in the next position, which may be the first position if no characters were extracted. operator>> then calls width(0).

If the function extracted no characters, it calls setstate(failbit), which may throw ios_base::failure (29.5.5.4).

Returns: in.

template<class charT, class traits>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT& c);

Effects: Behaves like a formatted input member (as described in 29.7.4.2.1) of in. After a sentry object is constructed a character is extracted from in, if one is available, and stored in c. Otherwise, the function calls in.setstate(failbit).

Returns: in.

basic_istream<charT, traits>& operator>>(basic_streambuf<charT, traits>* sb);

Effects: Behaves as an unformatted input function (29.7.4.3). If sb is null, calls setstate(failbit), which may throw ios_base::failure (29.5.5.4). After a sentry object is constructed, extracts characters from *this and inserts them in the output sequence controlled by sb. Characters are extracted and inserted until any of the following occurs:

1. end-of-file occurs on the input sequence;
2. inserting in the output sequence fails (in which case the character to be inserted is not extracted);
3. an exception occurs (in which case the exception is caught).

If the function inserts no characters, it calls setstate(failbit), which may throw ios_base::failure (29.5.5.4). If it inserted no characters because it caught an exception thrown while extracting...
characters from *this and failbit is on in exceptions() (29.5.5.4), then the caught exception is
rethrown.

Returns: *this.

### 29.7.4.3 Unformatted input functions

Each unformatted input function begins execution by constructing an object of class **sentry** with the default
argument **noskipws** (second) argument **true**. If the **sentry** object returns **true**, when converted to a value
of type **bool**, the function endeavors to obtain the requested input. Otherwise, if the **sentry** constructor exits
by throwing an exception or if the **sentry** object returns **false**, when converted to a value of type **bool**, the
function returns without attempting to obtain any input. In either case the number of extracted characters is
set to 0; unformatted input functions taking a character array of nonzero size as an argument shall also store
a null character (using **charT()**) in the first location of the array. If an exception is thrown during input then
**ios::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;
```

*Effects:* None. This member function does not behave as an unformatted input function (as described
above).

*Returns:* The number of characters extracted by the last unformatted input member function called for
the object.

```cpp
int_type get();
```

*Effects:* Behaves as an unformatted input function (as described above). After constructing a **sentry**
object, extracts a character c, if one is available. Otherwise, the function calls **setstate(failbit)**,
which may throw **ios_base::failure** (29.5.5.4),

*Returns:* c if available, otherwise **traits::eof()**.

```cpp
basic_istream<charT, traits>& get(char_type& c);
```

*Effects:* Behaves as an unformatted input function (as described above). After constructing a **sentry**
object, extracts a character, if one is available, and assigns it to c. Otherwise, the function calls
**setstate(failbit)** (which may throw **ios_base::failure** (29.5.5.4)).

*Returns:* *this.

```cpp
basic_istream<charT, traits>& get(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. Characters are extracted and stored until any of the following occurs:

- n is less than one or n - 1 characters are stored;
- end-of-file occurs on the input sequence (in which case the function calls **setstate(eofbit)**);
- **traits::eq(c, delim)** for the next available input character c (in which case c is not extracted).

If the function stores no characters, it calls **setstate(failbit)** (which may throw **ios_base::failure**
(29.5.5.4)). In any case, if n is greater than zero it then stores a null character into the next successive
location of the array.

*Returns:* *this.

```cpp
basic_istream<charT, traits>& get(char_type* s, streamsize n);
```

*Effects:* Calls get(s, n, widen(‘\n’)).

*Returns:* Value returned by the call.

---

310) This is done without causing an **ios::failure** to be thrown.

311) Note that this function is not overloaded on types **signed char** and **unsigned char**.

312) Note that this function is not overloaded on types **signed char** and **unsigned char**.
basic_istream<char, traits>& get(basic_streambuf<char_type, traits>& sb, char_type delim);

**Effects:** Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and inserts them in the output sequence controlled by sb. Characters are extracted and inserted until any of the following occurs:

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. traits::eq(c, delim) for the next available input character c (in which case c is not extracted);
4. an exception occurs (in which case, the exception is caught but not rethrown).

If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (29.5.5.4).

**Returns:** *this.

basic_istream<char, traits>& get(basic_streambuf<char_type, traits>& sb);

**Effects:** Calls `get(sb, widen('\n'))`.

**Returns:** Value returned by the call.

basic_istream<char, 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. 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);
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.

If the function extracts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (29.5.5.4)).

In any case, if n is greater than zero, it then stores a null character (using `charT()`) into the next successive location of the array.

**Returns:** *this.

[Example:

```cpp
#include <iostream>

int main() {
    using namespace std;
    const int line_buffer_size = 100;

    char buffer[line_buffer_size];
    int line_number = 0;
    while (cin.getline(buffer, line_buffer_size, '\n') || cin.gcount()) {
        int count = cin.gcount();
        if (cin.eof())
            cout << "Partial final line"; // cin.fail() is false
        else if (cin.fail()) {
            cout << "Partial long line";
            cin.clear(cin.rdstate() & ~ios_base::failbit);
        } else {
            count--;
                // Don't include newline in count
        }
    }
}
```

313) Note that this function is not overloaded on types `signed char` and `unsigned char`.
314) Since the final input character is “extracted”, it is counted in the `gcount()`, even though it is not stored.
315) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.
316) This implies an empty input line will not cause `failbit` to be set.
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’))

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 != numeric_limits<streamsize>::max() (17.3.4) and n characters have been extracted so
far
(25.2) end-of-file occurs on the input sequence (in which case the function calls setstate(eofbit),
which may throw ios_base::failure (29.5.5.4));
(25.3) traits::eq_int_type(traits::to_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 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: traits::eof() if good() is false. Otherwise, returns rdbuf()->sgetc().

basic_istream<charT, traits>& read(char_type* s, streamsize n);

Effects: Behaves as an unformatted input function (as described above). After constructing a sentry
object, if !good() calls setstate(failbit) which may throw an exception, and return. Otherwise
extracts characters and stores them into successive locations of an array whose first element is designated
by s.  Characters are extracted and stored until either of the following occurs:

(30.1) n characters are stored;
(30.2) end-of-file occurs on the input sequence (in which case the function calls setstate(failbit |
eofbit), which may throw ios_base::failure (29.5.5.4)).

Returns: *this.

streamsize readsome(char_type* s, streamsize n);

Effects: Behaves as an unformatted input function (as described above). After constructing a sentry
object, if !good() calls setstate(failbit) which may throw an exception, and return. Otherwise
extracts characters and stores them into successive locations of an array whose first element is designated
by s. If rdbuf() -> in_avail() == -1, calls setstate(eofbit) (which may throw
ios_base::failure (29.5.5.4)), and extracts no characters;

(32.1) If rdbuf() -> in_avail() == 0, extracts no characters
(32.2) If rdbuf() -> in_avail() > 0, extracts min(rdbuf() -> in_avail(), n)).

Returns: The number of characters extracted.

basic_istream<charT, traits>& putback(char_type c);

Effects: Behaves as an unformatted input function (as described above), except that the function first
clears eofbit. After constructing a sentry object, if !good() calls setstate(failbit) which may
throw an exception, and return. If rdbuf() is not null, calls rdbuf() -> sputbackc(c). If rdbuf()
is null, or if sputbackc returns traits::eof(), calls setstate(badbit) (which may throw ios_base::failure (29.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to gcount() is 0. — end note]

Returns: *this.

basic_istream<charT, traits>& unget();

Effects: Behaves as an unformatted input function (as described above), except that the function first clears eofbit. After constructing a sentry object, if !good() calls setstate(failbit) which may throw an exception, and return. If rdbuf() is not null, calls rdbuf()->sungetc(). If rdbuf() is null, or if sungetc returns traits::eof(), calls setstate(badbit) (which may throw ios_base::failure (29.5.5.4)). [Note: This function extracts no characters, so the value returned by the next call to gcount() is 0. — end note]

Returns: *this.

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.

pos_type tellg();

Effects: Behaves as an unformatted input function (as described above), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to gcount().

Returns: After constructing a sentry object, if fail() != false, returns pos_type(-1) to indicate failure. Otherwise, returns rdbuf()->pubseekoff(0, cur, in).

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).

Returns: *this.

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).

Returns: *this.

§ 29.7.4.4 Standard basic_istream manipulators [istream.manip]

Each instantiation of the function template specified in this subclause is a designated addressable function (16.5.4.2.1).

template<class charT, class traits>
basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);

Effects: Behaves as an unformatted input function (29.7.4.3), 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.

Returns: is.
29.7.4.5 Rvalue stream extraction

```
template<class charT, class traits, class T>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& is, T&& x);
```

1. **Effects:** Equivalent to:
   
   ```
   is >> std::forward<T>(x);
   return is;
   ```

2. **Remarks:** This function shall not participate in overload resolution unless the expression `is >> std::forward<T>(x)` is well-formed.

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& rhs) = delete;
  basic_iostream(basic_iostream&& rhs);

  // 29.7.4.6.3, assign and swap
  basic_iostream& operator=(const basic_iostream& rhs) = delete;
  basic_iostream& operator=(basic_iostream&& rhs);
  void swap(basic_iostream& rhs);
};
```

1. 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);
```

1. **Effects:** Constructs an object of class `basic_iostream`, initializing the base class subobjects with `basic_istream<charT, traits>(sb)` (29.7.4.1) and `basic_ostream<charT, traits>(sb)` (29.7.5.1).

2. **Ensures:** `rdbuf() == sb` and `gcount() == 0`.

29.7.4.6.2 Destructor

```
virtual ~basic_iostream();
```

1. **Effects:** Destroys an object of class `basic_iostream`.

2. **Remarks:** Does not perform any operations on `rdbuf()`.
29.7.4.6.3 Assignment and swap

basic_iostream& operator=(basic_iostream&& rhs);

Effects: As if by 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 values.

29.7.5.1 Class template basic_ostream

namespace std {

    template<class charT, class traits = char_traits<charT>>
    class basic_ostream : virtual public basic_ios<charT, traits> {

        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;

        explicit basic_ostream(basic_streambuf<char_type, traits>* sb);
        virtual ~basic_ostream();

        class sentry;

        basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&&));
        basic_ostream<charT, traits>& operator<<(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
        basic_ostream<charT, traits>& operator<<(ios_base& (*pf)(ios_base&));
        basic_ostream<charT, traits>& operator<<(bool n);
        basic_ostream<charT, traits>& operator<<(short n);
        basic_ostream<charT, traits>& operator<<(unsigned short n);
        basic_ostream<charT, traits>& operator<<(int n);
        basic_ostream<charT, traits>& operator<<(unsigned int n);
        basic_ostream<charT, traits>& operator<<(long n);
        basic_ostream<charT, traits>& operator<<(unsigned long n);
        basic_ostream<charT, traits>& operator<<(long long n);
        basic_ostream<charT, traits>& operator<<(unsigned long long n);
        basic_ostream<charT, traits>& operator<<(float f);
        basic_ostream<charT, traits>& operator<<(double f);
        basic_ostream<charT, traits>& operator<<(const void* p);
        basic_ostream<charT, traits>& operator<<(nullptr_t);
        basic_ostream<charT, traits>& operator<<(basic_streambuf<char_type, traits>* sb);

        // 29.7.5.3, 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();

    };

} // namespace std
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 error state. If `badbit` is on in `exceptions()`, the output function rethrows the exception without completing its actions, otherwise it does not throw anything and treat as an error.

### 29.7.5.1.1 Constructors

[ostream.cons]

```cpp
explicit basic_ostream(basic_streambuf<charT, traits>* sb);
```

**Effects:** Constructs an object of class `basic_ostream`, initializing the base class subobject with `basic_ios<charT, traits>::init(sb)` (29.5.5.2).

**Ensures:** `rdbuf() == sb`. 

---

1 The class template `basic_ostream` defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted output functions* (or *inserters*) and the *unformatted output functions*. Both groups of output functions generate (or *insert*) output characters by actions equivalent to calling `rdbuf()->sputc(int_type)`. They may use other public members of `basic_ostream` except that they shall not invoke any virtual members of `rdbuf()` except `overflow()`, `xsputn()`, and `sync()`.

3 If one of these called functions throws an exception, then unless explicitly noted otherwise the output function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the output function rethrows the exception without completing its actions, otherwise it does not throw anything and treat as an error.
basic_ostream(basic_ostream&& rhs);

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();

Effects: Destroys an object of class basic_ostream.

Remarks: Does not perform any operations on rdbuf().

29.7.5.1.2 Assignment and swap

basic_ostream& operator=(basic_ostream&& rhs);

Effects: As if by swap(rhs).

Returns: *this.

void swap(basic_ostream& rhs);

Effects: Calls basic_ios<charT, traits>::swap(rhs).

29.7.5.1.3 Class basic_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;

Effects: Returns ok_.

29.7.5.1.4 Seek members

Each seek member function begins execution by constructing an object of class sentry. It returns by destroying the sentry object.

pos_type tellp();

Returns: If fail() != false, returns pos_type(-1) to indicate failure. Otherwise, returns rdbuf()->pubseekoff(0, cur, out).

---

318) The call os.tie()->flush() does not necessarily occur if the function can determine that no synchronization is necessary.

319) The sentry constructor and destructor can also perform additional implementation-dependent operations.
basic_ostream<charT, traits>& seekp(pos_type pos);

**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);

**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

#### 29.7.5.2.1 Common requirements

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::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.

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

operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(unsigned long long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);

**Effects:** The classes num_get<> and num_put<> handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued locale value to perform numeric formatting. When val is of type bool, long, unsigned long, long long, unsigned long long, double, long double, or const void*, the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
    num_put<charT, ostreambuf_iterator<charT, traits>>
    >(getloc()).put(*this, *this, fill(), val).failed();
```

When val is of type short the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
```
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
>(getloc()).put(*this, *this, fill(),
  baseflags == ios_base::oct || baseflags == ios_base::hex
  ? static_cast<long>(static_cast<unsigned short>(val))
  : static_cast<long>(val)).failed();

When `val` is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
>(getloc()).put(*this, *this, fill(),
  baseflags == ios_base::oct || baseflags == ios_base::hex
  ? static_cast<long>(static_cast<unsigned int>(val))
  : static_cast<long>(val)).failed();
```

When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
>(getloc()).put(*this, *this, fill(),
  static_cast<unsigned long>(val)).failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```cpp
bool failed = use_facet<
  num_put<charT, ostreambuf_iterator<charT, traits>>
>(getloc()).put(*this, *this, fill(),
  static_cast<double>(val)).failed();
```

The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<` . It bypasses `ostreams` and uses `streambufs` directly. Class `locale` relies on these types as its interface to `iostreams`, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base class subobject of type `ios_base`. It provides formatting specifications such as field width, and a locale from which to obtain other facets. If `failed` is true then does `setstate(badbit)`, which may throw an exception, and returns.

Returns: `*this`.

### 29.7.5.2.3 basic_ostream::operator<<

`basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));`

1. **Effects:** None. Does not behave as a formatted output function (as described in 29.7.5.2.1).

Returns: `pf(*this)`.

2. **Effects:** Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 29.7.5.2.1).

Returns: `*this`.

3. **Effects:** Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 29.7.5.2.1).

Returns: `*this`.

---

321) See, for example, the function signature `endl(basic_ostream&)` (29.7.5.4).
322) See, for example, the function signature `dec(ios_base&)` (29.5.6.3).
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).

7 Gets characters from sb and inserts them in *this. Characters are read from sb and inserted until any of the following occurs:
8 — end-of-file occurs on the input sequence;
(8.1) — inserting in the output sequence fails (in which case the character to be inserted is not extracted);
(8.2) — an exception occurs while getting a character from sb.
(8.3) 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 error state, and if failbit is on in exceptions() the caught exception is rethrown.

9 Returns: *this.

basic_ostream<charT, traits>& operator<<(nullptr_t);

Effects: Equivalent to:
return *this << s;

where s is an implementation-defined NTCTS (16.3.17).

29.7.5.2.4 Character inserter function templates [ostream.inserters.character]

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, charT 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).

2 Returns: out.

template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const charT* s);

Requires: s shall not be 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*.
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

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::badbit is turned on\(^{323}\) 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.\(^{324}\) 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.\(^{325}\) 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 the function calls rdbuf() ->pubsync(). If that function returns -1 calls setstate(badbit) (which may throw ios_base::failure (29.5.5.4)). Otherwise, if the sentry object returns false, does nothing.

Returns: *this.

29.7.5.4 Standard manipulators

Each instantiation of any of the function templates specified in this subclause is a designated addressable function (16.5.4.2.1).

template<class charT, class traits>
  basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);

Effects: Calls os.put(os.widen(’\n’)), then os.flush().

Returns: os.

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(\0)).

Returns: os.

\(^{323}\) without causing an ios::failure to be thrown.

\(^{324}\) Note that this function is not overloaded on types signed char and unsigned char.

\(^{325}\) Note that this function is not overloaded on types signed char and unsigned char.
template<class charT, class traits>
    basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);

    Effects: Calls os.flush().
    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 basic_syncbuf<charT, traits, Allocator>* with a
             basic_syncbuf<charT, traits, Allocator>* that has no effect. [Note:
             To work around the Allocator template argument cannot be deduced, implementations
             can introduce an intermediate base class to basic_syncbuf that manages its emit_on_sync flag. — end
             note] Returns: os.

template<class charT, class traits>
    basic_ostream<charT, traits>& noemit_on_flush(basic_ostream<charT, traits>& os);

    Effects: If os.rdbuf() is a basic_syncbuf<charT, traits, Allocator>* with a
             basic_syncbuf<charT, traits, Allocator>* that 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 basic_syncbuf<charT, traits, Allocator>* with a
             basic_syncbuf<charT, traits, Allocator>* that has no effect.
    Returns: os.

29.7.5.5 Rvalue stream insertion  [ostream.rvalue]

    template<class charT, class traits, class T>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&& os, const T& x);

    Effects: As if by: os << x;
    Returns: os.
    Remarks: This function shall not participate in overload resolution unless the expression os << x is
             well-formed.

29.7.6 Standard manipulators  [std.manip]

The header <iomanip> defines several functions that support extractors and inserters that alter information
maintained by class ios_base and its derived classes.

    unspecified resetiosflags(ios_base::fmtflags mask);

    Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits>
             then the expression out << resetiosflags(mask) behaves as if it called f(out, mask), or if
             in is an object of type basic_istream<charT, traits> then the expression in >> resetiosflags(mask)
             behaves as if it called f(in, mask), where the function f is defined as:326

        void f(ios_base& str, ios_base::fmtflags mask) {
            // reset specified flags
            str.setf(ios_base::fmtflags(0), mask);
        }

    The expression out << resetiosflags(mask) shall have type basic_ostream<charT, traits>& and
    value out. The expression in >> resetiosflags(mask) shall have type basic_istream<charT, traits>& and
    value in.

326 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
unspecified setiosflags(ios_base::fmtflags mask);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out << setiosflags(mask) behaves as if it called f(out, mask), or if in is an object of type basic_istream<charT, traits> then the expression in >> setiosflags(mask) behaves as if it called f(in, mask), where the function f is defined as:

```cpp
void f(ios_base& str, ios_base::fmtflags mask) {
    // set specified flags
    str.setf(mask);
}
```

The expression out << setiosflags(mask) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setiosflags(mask) shall have type basic_istream<charT, traits>& and value in.

unspecified setbase(int base);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out << setbase(base) behaves as if it called f(out, base), or if in is an object of type basic_istream<charT, traits> then the expression in >> setbase(base) behaves as if it called f(in, base), where the function f is defined as:

```cpp
void f(ios_base& str, int base) {
    // set basefield
    str.setf(base == 8 ? ios_base::oct :
             base == 10 ? ios_base::dec :
             base == 16 ? ios_base::hex :
             ios_base::fmtflags(0), ios_base::basefield);
}
```

The expression out << setbase(base) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setbase(base) shall have type basic_istream<charT, traits>& and value in.

unspecified setfill(char_type c);

Returns: An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> and c has type charT then the expression out << setfill(c) behaves as if it called f(out, c), where the function f is defined as:

```cpp
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) shall have type basic_ostream<charT, traits>& and value out.

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) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setprecision(n) shall have type basic_istream<charT, traits>& and value in.

unspecified setw(int n);

Returns: An object of unspecified type such that if out is an instance of basic_ostream<charT, traits> then the expression out << setw(n) behaves as if it called f(out, n), or if in is an object
of type basic_istream<charT, traits> then the expression in >> setw(n) behaves as if it called f(in, n), where the function f is defined as:

```cpp
void f(ios_base& str, int n) {
    // set width
    str.width(n);
}
```

The expression out << setw(n) shall have type basic_ostream<charT, traits>& and value out. The expression in >> setw(n) shall have 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);

Requires: The type moneyT shall be 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) shall have type basic_istream<charT, traits>& and value in.

```cpp
template<class moneyT> unspecified put_money(const moneyT& mon, bool intl = false);

Requires: The type moneyT shall be either long double or a specialization of the basic_string template (Clause 21).

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_ios<charT, traits>& str, const moneyT& mon, bool Intl) {
    using Iter = ostreambuf_iterator<charT, traits>;
    using MoneyPut = money_put<charT, Iter>;
    const Iter end = mp.put(Iter(str.rdbuf()), Intl, str, str.fill(), mon);
    if (end.failed())
        str.setstate(ios::badbit);
}
```

The expression out << put_money(mon, Intl) shall have type basic_ostream<charT, traits>& and value out.
template<class charT> unspecified get_time(struct tm* tmb, const charT* fmt);

Requires: The argument tmb shall be a valid pointer to an object of type struct tm. The argument fmt shall be a valid pointer to an array of objects of type charT with char_traits<charT>::length(fmt) elements.

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_ios<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());
    err = 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) shall have type basic_istream<charT, traits>& and value in.

template<class charT> unspecified put_time(const struct tm* tmb, const charT* fmt);

Requires: The argument tmb shall be a valid pointer to an object of type struct tm, and the argument fmt shall be a valid pointer to an array of objects of type charT with char_traits<charT>::length(fmt) elements.

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_ios<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);
}
```

The expression out << put_time(tmb, fmt) shall have type basic_ostream<charT, traits>& and value out.

29.7.8 Quoted manipulators [quoted.manip]

[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]

```cpp
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) shall have 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.

The expression in >> quoted(s, delim, escape) shall have type basic_istream<charT, traits>& and value in. The expression out << quoted(s, delim, escape) shall have type basic_ostream<charT, traits>& and value out.

29.8  String-based streams

29.8.1  Header <sstream> synopsis
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>;

§ 29.8.1  1362
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
[stringbuf]

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>& str,
        ios_base::openmode which = ios_base::in | ios_base::out);
    basic_stringbuf(const basic_stringbuf& rhs) = delete;
    basic_stringbuf(basic_stringbuf&& rhs);

    // 29.8.2.2, assign and swap
    basic_stringbuf& operator=(const basic_stringbuf& rhs) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs);

    // 29.8.2.3, get and set
    basic_string<charT, traits, Allocator> str() const;
    void str(const 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;

§ 29.8.2
The class `basic_stringbuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary characters. The sequence can be initialized from, or made available as, an object of class `basic_string`.

For the sake of exposition, the maintained data is presented here as:

1. `ios_base::openmode mode`, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.

§ 29.8.2.1 Constructors

**[stringbuf.cons]**

1. `explicit basic_stringbuf(ios_base::openmode which);`

   **Effects:** Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (29.6.3.1), and initializing `mode` with `which`. It is implementation-defined whether the sequence pointers (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) are initialized to null pointers.

   **Ensures:** `str() == ""`.  

2. `explicit basic_stringbuf(const basic_string<charT, traits, Allocator>& s, ios_base::openmode which = ios_base::in | ios_base::out);`

   **Effects:** Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (29.6.3.1), and initializing `mode` with `which`. Then calls `str(s)`.  

3. `basic_stringbuf(basic_stringbuf&& rhs);`

   **Effects:** Move constructs from the rvalue `rhs`. 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. The openmode, locale and any other state of `rhs` is also copied.

   **Ensures:** 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.

   - `str() == rhs_p.str()`  
   - `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`  
   - `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`  
   - `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`  
   - `eptr() - pbase() == rhs_p.eptr() - rhs_p.pbase()`  
   - `if (eback()) eback() != rhs_a.eback()`  
   - `if (gptr()) gptr() != rhs_a.gptr()`  
   - `if (egptr()) egptr() != rhs_a.egptr()`  
   - `if (pbase()) pbase() != rhs_a.pbase()`  
   - `if (pptr()) pptr() != rhs_a.pptr()`  
   - `if (eptr()) epptr() != rhs_a.eptr()`
29.8.2.2 Assignment and swap

```cpp
data: 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
data: void swap(basic_stringbuf& rhs);
```

Effects: Exchanges the state of *this and rhs.

```cpp
data: template<class charT, class traits, class Allocator>
data: void swap(basic_stringbuf<charT, traits, Allocator>& x, 
data: basic_stringbuf<charT, traits, Allocator>& y);
```

Effects: As if by x.swap(y).

29.8.2.3 Member functions

```cpp
data: basic_string<charT, traits, Allocator> str() const;
```

Returns: A basic_string object whose content is equal to the basic_stringbuf underlying character sequence. If the basic_stringbuf was created only in input mode, the resultant basic_string contains the character sequence in the range [eback(), egptr()). If the basic_stringbuf was created with which & ios_base::out being nonzero then the resultant basic_string contains the character sequence in the range [pbase(), high_mark), 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 with a basic_string, or by calling the str(basic_string) member function. In the case of calling the str(basic_string) member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new basic_string). Otherwise the basic_stringbuf has been created in neither input nor output mode and a zero length basic_string is returned.

```cpp
data: void str(const basic_string<charT, traits, Allocator>& s);
```

Effects: Copies the content of s into the basic_stringbuf underlying character sequence and initializes the input and output sequences according to mode.

Ensures: If mode & ios_base::out is nonzero, pbase() points to the first underlying character and epptr() >= pbase() + s.size() holds; in addition, if mode & ios_base::ate is nonzero, pptr() == pbase() + s.size() holds, otherwise pptr() == pbase() is true. If mode & ios_base::in is nonzero, eback() points to the first underlying character, and both gptr() == eback() and egptr() == eback() + s.size() hold.

29.8.2.4 Overridden virtual functions

```cpp
data: 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.

```cpp
data: int_type pbackfail(int_type c = traits::eof()) override;
```

Effects: Puts back the character designated by c to the input sequence, if possible, in one of three ways:

1. If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if traits::eq(to_char_type(c), gptr()[-1]) returns true, assigns gptr() - 1 to gptr().

   Returns: c.

2. If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if mode & ios_base::out is nonzero, assigns c to *--gptr().

   Returns: c.

3. If traits::eq_int_type(c, traits::eof()) returns true and if the input sequence has a putback position available, assigns gptr() - 1 to gptr().
Returns: \texttt{traits::not_eof(c)}.

\textbf{Returns:} As specified above, or \texttt{traits::eof()} to indicate failure.

\textbf{Remarks:} If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

\begin{verbatim}
int_type overflow(int_type c = traits::eof()) override;
\end{verbatim}

\textbf{Effects:} Appends the character designated by \texttt{c} to the output sequence, if possible, in one of two ways:

\begin{itemize}
\item If \texttt{traits::eq_int_type(c, traits::eof())} returns \texttt{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 \texttt{sputc(c)}.
\item Signals success by returning \texttt{c}.
\item If \texttt{traits::eq_int_type(c, traits::eof())} returns \texttt{true}, there is no character to append.
\item Signals success by returning a value other than \texttt{traits::eof().}
\end{itemize}

\textbf{Remarks:} The function can alter the number of write positions available as a result of any call.

\textbf{Returns:} As specified above, or \texttt{traits::eof()} to indicate failure.

The function can make a write position available only if \texttt{(mode \& ios_base::out) \neq 0}. 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 \texttt{(mode \& ios_base::in) \neq 0}, the function alters the read end pointer \texttt{egptr()} to point just past the new write position.

\begin{verbatim}
pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out) override;
\end{verbatim}

\textbf{Effects:} Alters the stream position within one of the controlled sequences, if possible, as indicated in Table \ref{tab:seekoff-positioning}.

\begin{table}[ht]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Conditions} & \textbf{Result} \\
\hline
\texttt{(which \& ios_base::in) == ios_base::in} & positions the input sequence \\
\hline
\texttt{(which \& ios_base::out) == ios_base::out} & positions the output sequence \\
\hline
\texttt{(which \& (ios_base::in | ios_base::out)) == (ios_base::in | ios_base::out)} and either \texttt{way == ios_base::beg} or \texttt{way == ios_base::end} & positions both the input and the output sequences \\
\hline
\texttt{Otherwise} & the positioning operation fails. \\
\hline
\end{tabular}
\caption{seekoff positioning}
\end{table}

For a sequence to be positioned, the function determines \texttt{newoff} as indicated in Table \ref{tab:newoff-positioning}. If the sequence's next pointer (either \texttt{gpstr()} or \texttt{pptr()}) is a null pointer and \texttt{newoff} is nonzero, the positioning operation fails.

If \texttt{newoff + off} < 0, or if \texttt{newoff + off} refers to an uninitialized character (\ref{29.8.2.3}), the positioning operation fails. Otherwise, the function assigns \texttt{xbeg + newoff + off} to the next pointer \texttt{xnext}.

\textbf{Returns:} \texttt{pos_type(newoff)}, constructed from the resultant offset \texttt{newoff} (of type \texttt{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 \texttt{pos_type(off_type(-1))}.

\section*{§ 29.8.2.4}
### Table 113 — newoff values

<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>

pos_type seekpos(pos_type sp,  
    ios_base::openmode which  
      = ios_base::in | ios_base::out) override;

13 Effects: Equivalent to seekoff(off_type(sp), ios_base::beg, which).

14 Returns: `sp` to indicate success, or `pos_type(off_type(-1))` to indicate failure.

basic_streambuf<charT, traits>** setbuf(charT* s, streamsize n);

15 Effects: implementation-defined, except that setbuf(0, 0) has no effect.

16 Returns: `this`.

#### 29.8.3 Class template basic_istringstream

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>& str,  
            ios_base::openmode which = ios_base::in);
        basic_istringstream(const basic_istringstream& rhs) = delete;
        basic_istringstream(basic_istringstream&& rhs);

        // 29.8.3.2, assign and swap
        basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
        basic_istringstream& operator=(basic_istringstream&& rhs);
        void swap(basic_istringstream& rhs);

        // 29.8.3.3, members
        basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

        basic_string<charT, traits, Allocator> str() const;
        void str(const 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);  
    }

§ 29.8.3
The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the stringbuf object.

### 29.8.3.1 Constructors

```cpp
explicit basic_istringstream(ios_base::openmode which);
```

**Effects:** Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.1) and initializing sb with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)` (29.8.2.1).

```cpp
explicit basic_istringstream(const basic_string<charT, traits, Allocator>& str, ios_base::openmode which = ios_base::in);
```

**Effects:** Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(addressof(sb))` (29.7.4.1) and initializing sb with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)` (29.8.2.1).

```cpp
basic_istringstream(basic_istringstream&& rhs);
```

**Effects:** Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_stringbuf`.

### 29.8.3.2 Assignment and swap

```cpp
basic_istringstream& operator=(basic_istringstream&& rhs);
```

**Effects:** Move assigns the base and members of *this from the base and corresponding members of rhs.

**Returns:** *this.

```cpp
void swap(basic_istringstream& rhs);
```

**Effects:** Exchanges the state of *this and rhs by calling `basic_istream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```cpp
template<class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x, basic_istringstream<charT, traits, Allocator>& y);
```

**Effects:** As if by x.swap(y).

### 29.8.3.3 Member functions

```cpp
basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
```

**Returns:** `const_cast<basic_stringbuf<charT, traits, Allocator>*>(addressof(sb))`.

```cpp
basic_string<charT, traits, Allocator> str() const;
```

**Returns:** `rdbuf()->str()`.

```cpp
void str(const basic_string<charT, traits, Allocator>& s);
```

**Effects:** Calls `rdbuf()->str(s)`.

### 29.8.4 Class template basic_ostringstream

```cpp
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;
    }
}
```

§ 29.8.4 1368
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>& str,
    ios_base::openmode which = ios_base::out);
basic_ostringstream(const basic_ostringstream& rhs) = delete;
basic_ostringstream(basic_ostringstream&& rhs);

// 29.8.4.2, assign and swap
basic_ostringstream& operator=(const basic_ostringstream& rhs) = delete;
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;
void str(const basic_string<CharT, traits, Allocator>& s);
private:
basic_stringbuf<CharT, traits, Allocator> sb; // exposition only }

template<class CharT, class traits, class Allocator>
void swap(basic_ostringstream<CharT, traits, Allocator>& x,
        basic_ostringstream<CharT, traits, Allocator>& y);

1 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 [ostringstream.cons]

explicit basic_ostringstream(ios_base::openmode which);

1 Effects: Constructs an object of class basic_ostringstream<CharT, traits>, initializing the base class with basic_ostream<CharT, traits>(addressof(sb)) (29.7.5.1) and initializing sb with basic_stringbuf<CharT, traits, Allocator>(which | ios_base::out) (29.8.2.1).

explicit basic_ostringstream(
    const basic_string<CharT, traits, Allocator>& str,
    ios_base::openmode which = ios_base::out);

2 Effects: Constructs an object of class basic_ostringstream<CharT, traits>, initializing the base class with basic_ostream<CharT, traits>(addressof(sb)) (29.7.5.1) and initializing sb with basic_stringbuf<CharT, traits, Allocator>(str, which | ios_base::out) (29.8.2.1).

basic_ostringstream(basic_ostringstream& rhs);

3 Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Next basic_ostream<CharT, traits>::set_rdbuf(addressof(sb)) is called to install the contained basic_stringbuf.

29.8.4.2 Assignment and swap [ostringstream.assign]

basic_ostringstream& operator=(basic_ostringstream&& rhs);

1 Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.

2 Returns: *this.
void swap(basic_ostringstream& 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, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
          basic_ostringstream<charT, traits, Allocator>& y);

Effects: As if by 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;

Returns: rdbuf()->str().

void str(const basic_string<charT, traits, Allocator>& s);

Effects: Calls rdbuf()->str(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_iostream<charT, traits> { public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;
    using allocator_type = Allocator;

    // 29.8.5.1, constructors
    basic_stringstream() : basic_stringstream(ios_base::out | ios_base::in) {} // exposition only
    explicit basic_stringstream(ios_base::openmode which);
    explicit basic_stringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::out | ios_base::in);
    basic_stringstream(const basic_stringstream& rhs) = delete;
    basic_stringstream(basic_stringstream&& rhs);

    // 29.8.5.2, assign and swap
    basic_stringstream& operator=(const basic_stringstream& rhs) = delete;
    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;
    void str(const 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);
}
The class template `basic_stringstream<charT, traits>` supports reading and writing from objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

(1.1) — `sb`, the `stringbuf` object.

### 29.8.5.1 Constructors

**explicit basic_stringstream(ios_base::openmode which);**

1

**Effects:** Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which)`.

**explicit basic_stringstream(const basic_string<charT, traits, Allocator>& str, ios_base::openmode which = ios_base::out | ios_base::in);**

2

**Effects:** Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(addressof(sb))` (29.7.4.6.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

**basic_stringstream(basic_stringstream&& rhs);**

3

**Effects:** Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT, traits>::set_rdbuf(addressof(sb))` is called to install the contained `basic_stringbuf`.

### 29.8.5.2 Assignment and swap

**basic_stringstream& operator=(basic_stringstream&& rhs);**

1

**Effects:** Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

2

**Returns:** `*this`.

**void swap(basic_stringstream& rhs);**

3

**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, class Allocator>**

void swap(basic_stringstream<charT, traits, Allocator>& x, basic_stringstream<charT, traits, Allocator>& y);

4

**Effects:** As if by `x.swap(y)`.

### 29.8.5.3 Member functions

**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

**Returns:** `rdbuf()->str()`.

**void str(const basic_string<charT, traits, Allocator>& str);**

3

**Effects:** Calls `rdbuf()->str(str)`.

### 29.9 File-based streams

#### 29.9.1 Header `<fstream>` synopsis

```cpp
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class basic_filebuf;
  using filebuf = basic_filebuf<char>;
  using wfilebuf = basic_filebuf<wchar_t>;
} // namespace std
```
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 this subclause, 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` [filebuf]

```cpp
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>;
```
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);

The class basic_filebuf<charT, traits> associates both the input sequence and the output sequence with
a file.

The restrictions on reading and writing a sequence controlled by an object of class basic_filebuf<charT,
traits> are the same as for reading and writing with the C standard library FILEs.

In particular:
(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.

An instance of basic_filebuf behaves as described in 29.9.2 provided traits::pos_type is fpos<traits::
state_type>. Otherwise the behavior is undefined.

In order to support file I/O and multibyte/wide character conversion, conversions are performed using
members of a facet, referred to as a_codecvt in following subclauses, obtained as if by

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();

Effects: Constructs an object of class basic_filebuf<charT, traits>, initializing the base class with
basic_streambuf<charT, traits>() (29.6.3.1).

Ensures: is_open() == false.

basic_filebuf(basic_filebuf&& rhs);

Effects: Move constructs from the rvalue rhs. 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.

Ensures: 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()
virtual ~basic_filebuf();

Effects: Destroys an object of class basic_filebuf<charT, traits>. 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 16.5.5.12).

29.9.2.2 Assignment and swap

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: As if by x.swap(y).

29.9.2.3 Member functions

bool is_open() const;

Returns: true if a previous call to open succeeded (returned a non-null value) and there has been no intervening call to close.

basic_filebuf* open(const char* s, ios_base::openmode mode);

basic_filebuf* open(const filesystem::path::value_type* s, ios_base::openmode mode);

// wide systems only; see 29.9.1

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 114. If mode is not some combination of flags shown in the table then the open fails.

If the open operation succeeds and (mode & ios_base::ate) != 0, positions the file to the end (as if by calling fseek(file, 0, SEEK_END), where file is the pointer returned by calling fopen).\(^\text{327}\)

If the repositioning operation fails, calls close() and returns a null pointer to indicate failure.

Returns: this if successful, a null pointer otherwise.

basic_filebuf* open(const string& s, ios_base::openmode mode);

basic_filebuf* open(const filesystem::path& s, ios_base::openmode mode);

Returns: open(s.c_str(), mode);

\(^{327}\) 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).
basic_filebuf* close();

Effects: If is_open() == false, returns a null pointer. If a put area exists, calls overflow(traits::
eof()) to flush characters. If the last virtual member function called on *this (between underflow,
overflow, seekoff, and seekpos) was overflow then calls a_codecvt.unshift (possibly several
times) to determine a termination sequence, inserts those characters and calls overflow(traits::
eof()) again. Finally, regardless of whether any of the preceding calls fails or throws an exception,
the function closes the file (as if by calling fclose(file)). If any of the calls made by the function,
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.

Returns: this on success, a null pointer otherwise.

Ensures: is_open() == false.

29.9.2.4 Overridden virtual functions

streamsize showmanyc() override;

Effects: Behaves the same as basic_streambuf::showmanyc() (29.6.3.4).

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.

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:

```
char  extern_buf[XSIZE];
char*  extern_end;
charT intern_buf[ISIZE];
charT* intern_end;
codecvt_base::result r =
    a_codecvt.in(state, extern_buf, extern_buf+XSIZE, extern_end,
                   intern_buf, intern_buf+ISIZE, intern_end);
```
This shall be done in such a way that the class can recover the position (fpos_t) corresponding to each character between intern_buf and intern_end. If the value of r indicates that a_codecvt.in() ran out of space in intern_buf, retry with a larger intern_buf.

int_type uflow() override;

**Effects:** Behaves according to the description of basic_streambuf<charT, traits>::uflow(), with the specialization that a sequence of characters is read from the input 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:

(5.1) If traits::eq_int_type(c, traits::eof()) returns false and if the function makes a putback position available and if traits::eq(to_char_type(c), gptr()[-1]) returns true, decrements the next pointer for the input sequence, gptr().

Returns: c.

(5.2) If traits::eq_int_type(c, traits::eof()) returns false and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores c there.

Returns: c.

(5.3) If traits::eq_int_type(c, traits::eof()) returns true, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, gptr().

Returns: traits::not_eof(c).

**Returns:** As specified above, or traits::eof() to indicate failure.

**Remarks:** If is_open() == false, the function always fails.

The function does not put back a character directly to the input sequence.

If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

int_type overflow(int_type c = traits::eof()) override;

**Effects:** Behaves according to the description of basic_streambuf<charT, traits>::overflow(c), except that the behavior of “consuming characters” is performed by first converting as if by:

```cpp
char* b = pbase();
char* p = pptr();
char* end;

char xbuf[XSIZE];
char* xbuf_end;

codevt_base::result r =
a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

(10.1) If r == codecvt_base::error then fail.

(10.2) If r == codecvt_base::noconv then output characters from b up to (and not including) p.

(10.3) If r == codecvt_base::partial then output to the file characters from xbuf up to xbuf_end, and repeat using characters from end to p. If output fails, fail (without repeating).

(10.4) Otherwise output from xbuf to xbuf_end, and fail if output fails. At this point if b != p and b == end (xbuf isn’t large enough) then increase XSIZE and repeat from the beginning.

**Returns:** traits::not_eof(c) to indicate success, and traits::eof() to indicate failure. If is_-

open() == false, the function always fails.

basic_streambuf* setbuf(char_type* s, streamsize n) override;

**Effects:** If setbuf(0, 0) is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. “Unbuffered” means that pbase() and pptr() always return null and output to the file should appear as soon as possible.

§ 29.9.2.4 1376
pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which
    = ios_base::in | ios_base::out) override;

Effects: Let width denote a_codecvt.encoding(). If is_open() == false, or off != 0 & width <= 0, then the positioning operation fails. Otherwise, if way != basic_ios::cur or off != 0, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if width > 0, call fseek(file, width * off, whence), otherwise call fseek(file, 0, whence).

Remarks: “The last operation was output” means either the last virtual operation was overflow or the put buffer is non-empty. “Write any unshift sequence” means, if width if less than zero then call a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end) and output the resulting unshift sequence. The function determines one of three values for the argument whence, of type int, as indicated in Table 115.

Table 115 — seekoff effects

<table>
<thead>
<tr>
<th>way</th>
<th>Value</th>
<th>stdio Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic_ios::beg</td>
<td>SEEK_SET</td>
<td></td>
</tr>
<tr>
<td>basic_ios::cur</td>
<td>SEEK_CUR</td>
<td></td>
</tr>
<tr>
<td>basic_ios::end</td>
<td>SEEK_END</td>
<td></td>
</tr>
</tbody>
</table>

Returns: A newly constructed pos_type object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns pos_type(off_type(-1)).

pos_type seekpos(pos_type sp,
    ios_base::openmode which
    = ios_base::in | ios_base::out) override;

Alters the file position, if possible, to correspond to the position stored in sp (as described below). Altering the file position performs as follows:

1. if (om & ios_base::out) != 0, then update the output sequence and write any unshift sequence;
2. set the file position to sp as if by a call to fsetpos;
3. if (om & ios_base::in) != 0, then update the input sequence;

where om is the open mode passed to the last call to open(). The operation fails if is_open() returns false.

If sp is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If sp has not been obtained by a previous successful call to one of the positioning functions (seekoff or seekpos) on the same file the effects are undefined.

Returns: sp on success. Otherwise returns pos_type(off_type(-1)).

int sync() override;

Effects: If a put area exists, calls filebuf::overflow to write the characters to the file, then flushes the file as if by calling fflush(file). If a get area exists, the effect is implementation-defined.

void imbue(const locale& loc) override;

Requires: 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.

Effects: Causes characters inserted or extracted after this call to be converted according to loc until another call of imbue.

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& rhs) = delete;
            basic_ifstream(basic_ifstream&& rhs);

            // 29.9.3.2, assign and swap
            basic_ifstream& operator=(const basic_ifstream& rhs) = 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:

\( \text{sb} \), the filebuf object.

29.9.3.1 Constructors

basic_ifstream();
1
  Effects: Constructs an object of class basic_ifstream<charT, traits>, initializing the base class with basic_istream<charT, traits>(addressof(sb)) (29.7.4.1.1) and initializing 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: Constructs an object of class basic_ifstream<charT, traits>, initializing the base class
with basic_istream<charT, traits>(*addressof(sb)) (29.7.4.1.1) and initializing sb with basic_filebuf<charT, traits>() (29.9.2.1), then calls runbuf() -> 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: The same as basic_ifstream(s.c_str(), mode).

basic_ifstream(basic_ifstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_filebuf. Next basic_istream<charT, traits>::set_rdbuf(
   addressof(sb)) is called to install the contained basic_filebuf.

29.9.3.2 Assignment and swap

basic_ifstream& operator=(basic_ifstream&& rhs);

Effects: Move assigns the base and members of *this from the base and corresponding members
of rhs.

Returns: *this.

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: As if by x.swap(y).

29.9.3.3 Member functions

basic_filebuf<charT, traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT, traits>*>(addressof(sb)).

bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path::value_type* s,
   ios_base::openmode mode = ios_base::in); // wide systems only; see 29.9.1

Effects: Calls rdbuf() -> open(s, mode | ios_base::in). If that function does not return a null
pointer calls clear(), otherwise calls setstate(failbit) (which may throw ios_base::failure)
(29.5.5.4).

void open(const string& s, ios_base::openmode mode = ios_base::in);
void open(const filesystem::path& s, ios_base::openmode mode = ios_base::in);

Effects: Calls open(s.c_str(), mode).

void close();

Effects: Calls rdbuf() -> close() and, if that function returns a null pointer, calls setstate(failbit)
   (which may throw ios_base::failure) (29.5.5.4).
29.9.4 Class template basic_ofstream

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ofstream : public basic_ostream<charT, traits> {
    public:
        using char_type = charT;
        using int_type = typename traits::int_type;
        using pos_type = typename traits::pos_type;
        using off_type = typename traits::off_type;
        using traits_type = traits;

        // 29.9.4.1, 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,
                                ios_base::openmode mode = ios_base::out); // wide systems only; see 29.9.1
        explicit basic_ofstream(const string& s,
                                ios_base::openmode mode = ios_base::out);
        explicit basic_ofstream(const filesystem::path& s,
                                ios_base::openmode mode = ios_base::out);
        basic_ofstream(const basic_ofstream& rhs) = delete;
        basic_ofstream(basic_ofstream&& rhs);

        // 29.9.4.2, assign and swap
        basic_ofstream& operator=(const basic_ofstream& rhs) = 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,
                  ios_base::openmode mode = ios_base::out); // wide systems only; see 29.9.1
        void open(const string& s, ios_base::openmode mode = ios_base::out);
        void open(const filesystem::path& s, ios_base::openmode mode = ios_base::out);
        void close();
    private:
        basic_filebuf<charT, traits> sb; // exposition only
    };

    template<class charT, class traits>
    void swap(basic_ofstream<charT, traits>& x,
              basic_ofstream<charT, traits>& y);
}

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:

— sb, the filebuf object.

29.9.4.1 Constructors

basic_ofstream();

Effects: Constructs an object of class basic_ofstream<charT, traits>, initializing the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1.1) and initializing sb with basic_filebuf<charT, traits>(). (29.9.2.1).

explicit basic_ofstream(const char* s,
                        ios_base::openmode mode = ios_base::out);
explicit basic_ofstream(const filesystem::path::value_type* s,
    ios_base::openmode mode = ios_base::out);  // wide systems only; see 29.9.1

Effects: Constructs an object of class basic_ofstream<charT, traits>, initializing the base class with basic_ostream<charT, traits>(addressof(sb)) (29.7.5.1.1) and initializing 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: The same as basic_ofstream(s.c_str(), mode).

basic_ofstream(basic_ofstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_filebuf. Next basic_ostream<charT, traits>::set_rdbuf(addressof(sb)) is called to install the contained basic_filebuf.

29.9.4.2 Assignment and swap

basic_ofstream& operator=(basic_ofstream&& rhs);

Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.

Returns: *this.

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: As if by x.swap(y).

29.9.4.3 Member functions

basic_filebuf<charT, traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT, traits>*>(addressof(sb)).

bool is_open() const;

Returns: rdbuf()->is_open().

void open(const char* s, ios_base::openmode mode = ios_base::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

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;

            // 29.9.5.1, constructors
            basic_fstream();
            explicit basic_fstream(  
                const char* s,
                ios_base::openmode mode = ios_base::in | ios_base::out);
            explicit basic_fstream(  
                const filesystem::path::value_type* s,
                ios_base::openmode mode = ios_base::in|ios_base::out);  // wide systems only; see 29.9.1
            explicit basic_fstream(  
                const string& s,
                ios_base::openmode mode = ios_base::in | ios_base::out);
            explicit basic_fstream(  
                const filesystem::path& s,
                ios_base::openmode mode = ios_base::in | ios_base::out);
            basic_fstream(const basic_fstream& rhs) = delete;
            basic_fstream(basic_fstream&& rhs);

            // 29.9.5.2, assign and swap
            basic_fstream& operator=(const basic_fstream& rhs) = delete;
            basic_fstream& operator=(basic_fstream&& rhs);
            void swap(basic_fstream& rhs);

            // 29.9.5.3, members
            basic_filebuf<charT, traits>* rdbuf() const;
            bool is_open() const;
            void open(  
                const char* s,
                ios_base::openmode mode = ios_base::in | ios_base::out);
            void open(  
                const filesystem::path::value_type* s,
                ios_base::openmode mode = ios_base::in|ios_base::out);  // wide systems only; see 29.9.1
            void open(  
                const string& s,
                ios_base::openmode mode = ios_base::in | ios_base::out);
            void open(  
                const filesystem::path& s,
                ios_base::openmode mode = ios_base::in | ios_base::out);
            void close();

            private:
                basic_filebuf<charT, traits> sb;  // exposition only
            }

    template<class charT, class traits>
        void swap(basic_fstream<charT, traits>& x,
                basic_fstream<charT, traits>& y);
    }

1 The class template basic_fstream<charT, traits> supports reading and writing from named files. It uses a basic_filebuf<charT, traits> object to control the associated sequences. For the sake of exposition, the maintained data is presented here as:

(1.1) sb, the basic_filebuf object.


29.9.5.1 Constructors

basic_fstream();

Effects: Constructs an object of class basic_fstream<charT, traits>, initializing the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) and initializing sb with basic_filebuf<charT, traits>().

explicit basic_fstream(const char* s,
ios_base::openmode mode = ios_base::in | ios_base::out);

Parameters: s: the file name.

Effects: Constructs an object of class basic_fstream<charT, traits>, initializing the base class with basic_iostream<charT, traits>(addressof(sb)) (29.7.4.6.1) and initializing sb with basic_filebuf<charT, traits>().

Parameters: s: the file name.

29.9.5.2 Assignment and swap

basic_fstream& operator=(basic_fstream&& rhs);

Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.

Returns: *this.

void swap(basic_fstream& rhs);

Effects: Exchanges the state of *this and rhs by calling basic_iostream<charT, traits>::swap(rhs) and sb.swap(rhs.sb).

29.9.5.3 Member functions

basic_filebuf<charT, traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT, traits>*>(addressof(sb)).

bool is_open() const;

Returns: rdbuf()->is_open().
void open(
    const 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 \texttt{<syncstream>} synopsis

\texttt{#include <ostream>  // see 29.7.2}

namespace std {
    template<class charT, class traits, class Allocator>
    class basic_syncbuf;

    using syncbuf = basic_syncbuf<char>;
    using wsyncbuf = basic_syncbuf<wchar_t>;

    template<class charT, class traits, class Allocator>
    class basic_osyncstream;

    using osyncstream = basic_osyncstream<char>;
    using wosyncstream = basic_osyncstream<wchar_t>;
}

1 The header \texttt{<syncstream>} provides a mechanism to synchronize execution agents writing to the same stream.

29.10.2  Class template \texttt{basic_syncbuf}

29.10.2.1  Overview

namespace std {
    template<class charT, class traits, class Allocator>
    class basic_syncbuf : 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;

            using streambuf_type = basic_streambuf<charT, traits>;

            // 29.10.2.2, construction and destruction
            explicit basic_syncbuf(streambuf_type* obuf = nullptr)
            : basic_syncbuf(obuf, Allocator()) {}
            basic_syncbuf(streambuf_type*, const Allocator&);
            basic_syncbuf(basic_syncbuf&&);
            ~basic_syncbuf();
    }
}
// 29.10.2.2, construction and destruction
basic_syncbuf(streambuf_type* obuf, const Allocator& allocator);

Effects: Constructs the basic_syncbuf object and sets wrapped to obuf.

Remarks: A copy of allocator is used to allocate memory for internal buffers holding the associated output.

Throws: Nothing unless an exception is thrown by the construction of a mutex or by memory allocation.

Ensures: get_wrapped() == obuf and get_allocator() == allocator are true.

basic_syncbuf(basic_syncbuf&& other);

Effects: Move constructs from other (Table 26).

Ensures: The value returned by this->get_wrapped() is the value returned by other.get_wrapped() prior to calling this constructor. Output stored in other prior to calling this constructor will be stored in *this afterwards. other.rdbuf()->pbase() == other.rdbuf()->pptr() and other.get_wrapped() == nullptr are true. 

Remarks: This constructor disassociates other from its wrapped stream buffer, ensuring destruction of other produces no output.

basic_syncbuf& operator=(basic_syncbuf&& rhs) noexcept;

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).
Returns: *this.

Ensures:

(3.1) rhs.get_wrapped() == nullptr is true.
(3.2) this->get_allocator() == rhs.get_allocator() is true when
    allocator_traits<Allocator>::propagate_on_container_move_assignment::value
    is true; otherwise, the allocator is unchanged.

Remarks: This assignment operator disassociates rhs from its wrapped stream buffer, ensuring
destruction of rhs produces no output.

void swap(basic_syncbuf& other) noexcept;

Requires: Either allocator_traits<Allocator>::propagate_on_container_swap::value is true
or this->get_allocator() == other.get_allocator() is true.

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:

(2.1) wrapped == nullptr is false.
(2.2) All of the characters in the associated output were successfully transferred.
(2.3) The call to wrapped->pubsync() (if any) succeeded.

Ensures: 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 (6.8.2.1), 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

```cpp
namespace std {
    template<class charT, class traits, class Allocator>
    class basic_osyncstream : public basic_ostream<charT, traits> {
    public:
        using char_type = charT;
        using int_type = typename traits::int_type;
        using pos_type = typename traits::pos_type;
        using off_type = typename traits::off_type;
        using traits_type = traits;
        using allocator_type = Allocator;
        using streambuf_type = basic_streambuf<charT, traits>;
        using syncbuf_type = basic_syncbuf<charT, traits, Allocator>;

        basic_osyncstream(streambuf_type* buf, const Allocator& allocator);
        explicit basic_osyncstream(streambuf_type* obuf)
            : basic_osyncstream(obuf, Allocator()) {}
        basic_osyncstream(basic_ostream<charT, traits>& os, const Allocator& allocator)
            : basic_osyncstream(os.rdbuf(), allocator) {}
        explicit basic_osyncstream(basic_ostream<charT, traits>&& os)
            : basic_osyncstream(os, Allocator()) {}
        basic_osyncstream(basic_osyncstream&&) noexcept;
        ~basic_osyncstream();

        basic_osyncstream& operator=(basic_osyncstream&&) noexcept;

    private:
        syncbuf_type sb;  // exposition only
    };
}
```

1 Allocator shall satisfy the Cpp17Allocator requirements (Table 34).

2 [Example: A named variable can be used within a block statement for streaming.]

```cpp
osyncstream bout(cout);
bout << "Hello, ";
bout << "World!";
bout << endl;  // flush is noted
bout << "and more!\n";
// characters are transferred and cout is flushed
```

3 [Example: A temporary object can be used for streaming within a single statement.

```cpp
osyncstream(cout) << "Hello, " << "World!" << '\n';
```

In this example, cout is not flushed. —end example]

29.10.3.2 Construction and destruction

```cpp
basic_osyncstream(streambuf_type* buf, const Allocator& allocator);
```

1 Effects: Initializes sb from buf and allocator. Initializes the base class with basic_ostream<charT, traits>(addressof(sb)).
2 [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]

3 Ensures: get_wrapped() == buf is true.

basic_osyncstream(basic_osyncstream&& other) noexcept;

4 Effects: Move constructs the base class and sb from the corresponding subobjects of other, and calls basic_ostream<charT, traits>::set_rdbuf(addressof(sb)).

5 Ensures: 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.

~basic_osyncstream();

6 Effects: Calls emit(). If an exception is thrown from emit(), that exception is caught and ignored.

29.10.3.3 Assignment [[syncstream.osyncstream.assign]]

basic_osyncstream& operator=(basic_osyncstream&& rhs) noexcept;

1 Effects: First, calls emit(). If an exception is thrown from emit(), that exception is caught and ignored. Move assigns sb from rhs.sb. [Note: This disassociates rhs from its wrapped stream buffer ensuring destruction of rhs produces no output. — end note]

2 Ensures: nullptr == rhs.get_wrapped() is true. get_wrapped() returns the value previously returned by rhs.get_wrapped().

29.10.3.4 Member functions [[syncstream.osyncstream.members]]

void emit();

1 Effects: Calls sb.emit(). If that call returns false, calls setstate(ios::badbit).

2 [Example: A flush on a basic_osyncstream does not flush immediately:

   
   {  
     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]

3 [Example: The function emit() can be used to handle exceptions from operations on the underlying stream.

   
   {  
     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,

   
   {  
     osyncstream bout1(cout);  
     bout1 << "Hello, ";
   }
produces the uninterleaved output

Goodbye, Planet!
Hello, World!

— 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 this subclause 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, Requires: 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 This subclause mentions commercially available operating systems for purposes of exposition. 328

29.11.4 Requirements [fs.req]

1 Throughout this subclause, 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 satisfy the Cpp17InputIterator requirements (23.3.5.2) and shall have a value type that is one of the encoded character types.

4 [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]

5 Template parameters named Allocator shall satisfy the Cpp17Allocator requirements (Table 34).

29.11.4.1 Namespaces and headers [fs.req.namespace]

1 Unless otherwise specified, references to entities described in this subclause are assumed to be qualified with ::std::filesystem::.

29.11.5 Header <filesystem> synopsis [fs.filesystem.syn]

```cpp
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;
}
```

328  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.
// 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;
};

// 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 option);
bool copy_file(const path& from, const path& to, copy_options option,
               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);
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;

space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;

file_status status(const path& p);
file_status status(const path& p, error_code& ec) noexcept;

bool status_known(file_status s) noexcept;

file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;

path temp_directory_path();
path temp_directory_path(error_code& ec);

path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);

1 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 [fs.err.report]

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:

(1.1) — Uses where file system errors are truly exceptional and indicate a serious failure. Throwing an exception is an appropriate response.

(1.2) — Uses where file system errors are routine and do not necessarily represent failure. Returning an error code is the most appropriate response. This allows application specific error handling, including simply ignoring the error.
Functions not having an argument of type `error_code&` handle errors as follows, unless otherwise specified:

(2.1) When a call by the implementation to an operating system or other underlying API results in an error that prevents the function from meeting its specifications, an exception of type `filesystem_error` shall be thrown. For functions with a single path argument, that argument shall be passed to the `filesystem_error` constructor with a single path argument. For functions with two path arguments, the first of these arguments shall be passed to the `filesystem_error` constructor as the `path1` argument, and the second shall be passed as the `path2` argument. The `filesystem_error` constructor's `error_code` argument is set as appropriate for the specific operating system dependent error.

(2.2) Failure to allocate storage is reported by throwing an exception as described in 16.5.5.12.

(2.3) Destructors throw nothing.

Functions having an argument of type `error_code&` handle errors as follows, unless otherwise specified:

(3.1) 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

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.

[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]
template<class Source>
    path(const Source& source, format fmt = auto_format);
template<class InputIterator>
    path(InputIterator first, InputIterator last, format fmt = auto_format);
template<class 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);
    ~path();

    // 29.11.7.4.2, assignments
    path& operator=(const path& p);
    path& operator=(path&& p) noexcept;
    path& operator=(string_type&& source);
    template<class Source>
        path& operator=(const Source& source);
    template<class Source>
        path& assign(const Source& source);
template<class InputIterator>
    path& operator=(InputIterator first, InputIterator last);

template<class Source>
    path& operator=(string_type&& source);
    path& assign(string_type&& source);
    template<class Source>
        path& assign(const Source& source);
    template<class InputIterator>
        path& assign(InputIterator first, InputIterator last);

template<class Source>
    path& operator=(const Source& source);
    template<class Source>
        path& assign(const Source& source);
template<class InputIterator>
    path& assign(InputIterator first, InputIterator last);

    // 29.11.7.4.3, 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);

    // 29.11.7.4.4, concatenation
    path& operator+=(const path& x);
    path& operator+=(const string_type& x);
    path& operator+=(basic_string_view<value_type> x);
    path& operator+=(const value_type* x);
    path& operator+=(value_type x);
    template<class Source>
        path& operator+=(const Source& x);
    template<class EcharT>
        path& operator+=(EcharT x);
    template<class Source>
        path& concat(const Source& x);
    template<class InputIterator>
        path& concat(InputIterator first, InputIterator last);

    // 29.11.7.4.5, modifiers
    void clear() noexcept;
    path& make_preferred();
    path& remove_filename();
    path& replace_filename(const path& replacement);
    path& replace_extension(const path& replacement = path());
    void swap(path& rhs) noexcept;

    // 29.11.7.7, non-member operators
    friend bool operator==(const path& lhs, const path& rhs) noexcept;
    friend bool operator!=(const path& lhs, const path& rhs) noexcept;
    friend bool operator<( const path& lhs, const path& rhs) noexcept;
    friend bool operator<=(const path& lhs, const path& rhs) noexcept;
    friend bool operator>( const path& lhs, const path& rhs) noexcept;
    friend bool operator>=(const path& lhs, const path& rhs) noexcept;
    friend path operator/ (const path& lhs, const path& rhs);
// 29.11.7.4.6, native format observers
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;

// 29.11.7.4.7, generic format observers
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;

// 29.11.7.4.8, compare
int compare(const path& p) const noexcept;
int compare(const string_type& s) const;
int compare(basic_string_view<value_type> s) const;
int compare(const value_type* s) const;

// 29.11.7.4.9, decomposition
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>
template<class charT, class traits>
friend basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const path& p);

// template<class charT, class traits>
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
  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 /

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:

— 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]

— The maximum permitted length.

— Filenames that are not permitted.

— Filenames that have special meaning.

— Case awareness and sensitivity during path resolution.

— Special rules that may apply to file types other than regular files, such as directories.

§ 29.11.7.1
Except in a root-name, multiple successive directory-separator characters are considered to be the same as one directory-separator character.

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.

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]

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]

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.

### Conversions

Argument format conversions [fs.path.fmt.cvt]

1. 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]

2. 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.

3. [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]

4. 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]

5. 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.
6 [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]

7 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).

8 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  

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:

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. 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]

3. char8_t: The encoding is UTF-8. The method of conversion is unspecified.

4. char16_t: The encoding is UTF-16. The method of conversion is unspecified.

5. char32_t: The encoding is UTF-32. The method of conversion is unspecified.

If the encoding being converted to has no representation for source characters, the resulting converted characters, if any, are unspecified. Implementations should not modify member function arguments if already of type path::value_type.

29.11.7.3 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()).

2. basic_string_view<EcharT, traits>. A function argument const Source& source shall have an effective range [source.begin(), source.end()).

3. A type meeting the Cpp17InputIterator requirements that iterates over a NTCTS. The value type shall be an encoded character type. A function argument const Source& source shall have an effective range [source, end) where end is the first iterator value with an element value equal to iterator_traits<Source>::value_type().

4. A character array that after array-to-pointer decay results in a pointer to the start of a NTCTS. The value type shall be an encoded character type. A function argument const Source& source shall have an effective range [source, end) where end is the first iterator value with an element value equal to iterator_traits<decay_t<Source>>::value_type().

Functions taking template parameters named Source shall not participate in overload resolution unless either

1. Source is a specialization of basic_string or basic_string_view, or

2. the qualified-id iterator_traits<decay_t<Source>>::value_type is valid and denotes a possibly const encoded character type (13.9.2).
Arguments of type Source shall not be null pointers.

29.11.7.4 Members

29.11.7.4.1 Constructors

path() noexcept;
  Effects: Constructs an object of class path.
  Ensures: empty() == true.

path(const path& p);
path(path&& p) noexcept;
  Effects: Constructs an object of class path having the same pathname in the native and generic formats, respectively, as the original value of p. In the second form, p is left in a valid but unspecified state.

path(string_type&& source, format fmt = auto_format);
  Effects: Constructs an object of class path for which the pathname in the detected-format of source has the original value of source (29.11.7.2.1), 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);

6  Requires: The value type of Source and InputIterator is char.

7  Effects: Let s be the effective range of source or the range [first, last), after converting the encoding as follows:
   (7.1) 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.
   (7.2) 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.

8  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:
  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 ordinary encoding. If the native wide encoding is UTF-16 or UTF-32, and the current native ordinary encoding is UTF-8, all of the characters in the ISO/IEC 8859-1 character set will be converted to their Unicode representation, but for other native ordinary encodings some characters may have no representation.
For Windows-based operating systems, the path is constructed by using \texttt{latin1\_facet} to convert ISO/IEC 8859-1 encoded \texttt{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

\texttt{path\& operator=(const path\& p);}  
\begin{enumerate}
\item \textbf{Effects:} If \texttt{*this} and \texttt{p} are the same object, has no effect. Otherwise, sets both respective pathnames of \texttt{*this} to the respective pathnames of \texttt{p}.
\item \textbf{Returns:} \texttt{*this}.
\end{enumerate}

\texttt{path\& operator=(path\& p) noexcept;}  
\begin{enumerate}
\item \textbf{Effects:} If \texttt{*this} and \texttt{p} are the same object, has no effect. Otherwise, sets both respective pathnames of \texttt{*this} to the respective pathnames of \texttt{p}. \texttt{p} is left in a valid but unspecified state. \textit{[Note: A valid implementation is \texttt{swap(p)}.} — \textit{end note]}
\item \textbf{Returns:} \texttt{*this}.
\end{enumerate}

\texttt{path\& operator=(string\_type\&\& source);}  
\texttt{path\& assign(string\_type\&\& source);}  
\begin{enumerate}
\item \textbf{Effects:} Sets the pathname in the detected-format of \texttt{source} to the original value of \texttt{source}. \texttt{source} is left in a valid but unspecified state.
\item \textbf{Returns:} \texttt{*this}.
\end{enumerate}

\texttt{template<class Source> \newline path\& operator=(const Source\& source);}  
\texttt{template<class Source> \newline path\& assign(const Source\& source);}  
\texttt{template<class InputIterator> \newline path\& assign(InputIterator first, InputIterator last);}  
\begin{enumerate}
\item \textbf{Effects:} Let \texttt{s} be the effective range of \texttt{source} (29.11.7.3) or the range \texttt{[first, last)}, with the encoding converted if required (29.11.7.2). Finds the detected-format of \texttt{s} (29.11.7.2.1) and sets the pathname in that format to \texttt{s}.
\item \textbf{Returns:} \texttt{*this}.
\end{enumerate}

29.11.7.4.3 Appends

The append operations use \texttt{operator/=} to denote their semantic effect of appending \texttt{preferred-separator} when needed.

\texttt{path\& operator/=\!(const path\& p);}  
\begin{enumerate}
\item \textbf{Effects:} If \texttt{p.is\_absolute()} \texttt{|| (p.has\_root\_name() && p.root\_name() \!\!= \texttt{root}\_name())}, then \texttt{operator\!\!=(p)}.
\item Otherwise, modifies \texttt{*this} as if by these steps:
\begin{enumerate}
\item If \texttt{p.has\_root\_directory()}, then removes any root directory and relative path from the generic format pathname. Otherwise, if \texttt{!has\_root\_directory()} \texttt{&& is\_absolute()} is \texttt{true} or if \texttt{has\_-filename()} is \texttt{true}, then appends \texttt{path::preferred\_separator} to the generic format pathname.
\item Then appends the native format pathname of \texttt{p}, omitting any \texttt{root\_name} from its generic format pathname, to the native format pathname.
\end{enumerate}
\item [\textit{Example:} Even if \texttt{/host} is interpreted as a \texttt{root\_name}, both of the paths \texttt{path("/host")/"foo"} and \texttt{path("/host")/"foo"} equal \texttt{"/host/foo"}.

Expression examples:

\texttt{// On POSIX, \newline path("foo") / "}; \texttt{// yields "foo/" \newline path("foo") / "/bar";} \texttt{// yields "/bar" \newline // On Windows, backslashes replace slashes in the above yields}

\§ 29.11.7.4.3
On Windows,

```cpp
path("foo") / "c:/bar"; // yields "c:/bar"
path("foo") / "c:"; // yields "c:"
path("c:" / ""; // yields "c:"
path("c:foo") / "/bar"; // yields "c:/bar"
path("c:foo") / "c:bar"; // yields "c:foo/bar"
```

— end example

5 Returns: *this.

```cpp
template<class Source>
path& operator/=(const Source& source);
template<class Source>
path& append(const Source& source);
```

6 Effects: Equivalent to: return operator/=(path(source));

```cpp
template<class InputIterator>
path& append(InputIterator first, InputIterator last);
```

7 Effects: Equivalent to: return operator/=(path(first, last));

### 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);
path& operator+=(value_type x);
template<class Source>
path& operator+=(const Source& x);
template<class EcharT>
path& operator+=(EcharT x);
template<class Source>
path& concat(const Source& x);
```

1 Effects: Appends path(x).native() to the pathname in the native format. [Note: This directly manipulates the value of native() and may not be portable between operating systems. — end note]

2 Returns: *this.

```cpp
template<class InputIterator>
path& concat(InputIterator first, InputIterator last);
```

3 Effects: Equivalent to: return *this += path(first, last);

### 29.11.7.4.5 Modifiers

```cpp
void clear() noexcept;
```

1 Ensures: empty() == true.

```cpp
path& make_preferred();
```

2 Effects: Each directory-separator of the pathname in the generic format is converted to preferred-separator.

3 Returns: *this.

4 [Example:

```cpp
path p("foo/bar");
std::cout << p << 
'\n';
p.make_preferred();
std::cout << p << 
'\n';
```

On an operating system where preferred-separator is a slash, the output is:

"foo/bar"
"foo/bar"

On an operating system where preferred-separator is a backslash, the output is:
path& remove_filename();

Ensures: !has_filename().
Effects: Remove the generic format pathname of filename() from the generic format pathname.
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/bar").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:
— Any existing extension() (29.11.7.4.9) is removed from the pathname in the generic format, then
— 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
— 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 [fs.path.native.obs]
The string returned by all native format observers is in the native pathname format (29.11.7).

const string_type& native() const noexcept;

Returns: The pathname in the native format.

const value_type* c_str() const noexcept;

Effects: Equivalent to: return native().c_str();
operator string_type() const;

Returns: native().

[Note: Conversion to string_type is provided so that an object of class path can be given as an
argument to existing standard library file stream constructors and open functions. — end note]
template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
basic_string<EcharT, traits, Allocator>
    string(const Allocator& a = Allocator()) const;

Returns: native().
Remarks: All memory allocation, including for the return value, shall be performed by a. Conversion, if any, is specified by 29.11.7.2.

std::string string() const;
std::wstring wstring() const;
std::u8string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;

Returns: native().
Remarks: Conversion, if any, is performed as specified by 29.11.7.2.

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.

[Example: On an operating system that uses backslash as its preferred-separator,

    path("foo\bar").generic_string()

returns "foo/bar". — end example]

template<class EcharT, class traits = char_traits<EcharT>,
class Allocator = allocator<EcharT>>
    basic_string<EcharT, traits, Allocator>
    generic_string(const Allocator& a = Allocator()) const;

Returns: The pathname in the generic format.
Remarks: All memory allocation, including for the return value, shall be performed by a. Conversion, if any, is specified by 29.11.7.2.

std::string generic_string() const;
std::wstring generic_wstring() const;
std::u8string generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;

Returns: The pathname in the generic format.
Remarks: Conversion, if any, is specified by 29.11.7.2.

29.11.7.4.8 Compare

int compare(const path& p) const noexcept;

Returns:
(1.1) — Let rootNameComparison be the result of this->root_name().native().compare(p.root_name().native()). If rootNameComparison is not 0, rootNameComparison.
(1.2) — Otherwise, if !this->has_root_directory() and !p.has_root_directory(), a value less than 0.
(1.3) — Otherwise, if this->has_root_directory() and p.has_root_directory(), a value greater than 0.
(1.4) — Otherwise, if native() for the elements of this->relative_path() are lexicographically less than native() for the elements of p.relative_path(), a value less than 0.
(1.5) — Otherwise, if native() for the elements of this->relative_path() are lexicographically greater than native() for the elements of p.relative_path(), a value greater than 0.
(1.6) — Otherwise, 0.
int compare(const string_type& s) const
int compare(basic_string_view<value_type> s) const;
int compare(const value_type* s) const

Effects: Equivalent to: return compare(path(s));

29.11.7.4.9 Decomposition

path root_name() const;
1
Returns: root-name, if the pathname in the generic format includes root-name, otherwise path().

path root_directory() const;
2
Returns: root-directory, if the pathname in the generic format includes root-directory, otherwise path().

path root_path() const;
3
Returns: root_name() / root_directory().

path relative_path() const;
4
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;
5
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;
6
Returns: relative_path().empty() ? path() : *--end().

[Example:
  path("/foo/bar.txt").filename(); // yields "bar.txt"
  path("/foo/bar").filename(); // yields "bar"
  path("/foo/bar/").filename(); // yields 
  path("/").filename(); // yields 
  path("//host").filename(); // yields 
  path(".").filename(); // yields 
  path("..").filename(); // yields 
  — end example]

path stem() const;
8
Returns: Let f be the generic format pathname of filename(). Returns a path whose pathname in the generic format is

(8.1) f, if it contains no periods other than a leading period or consists solely of one or two periods;

(8.2) otherwise, the prefix of f ending before its last period.

[Example:
  std::cout << path("/foo/bar.txt").stem(); // outputs "bar"
  path p = "foo.bar.baz.tar";
  for (; !p.extension().empty(); p = p.stem())
    std::cout << p.extension() << \\
    // outputs: .tar
    // .baz
    // .bar
    — end example]

path extension() const;
10
Returns: A path whose pathname in the generic format is the suffix of filename() not included in stem().

[Example:
path("/foo/bar.txt").extension(); // yields ".txt" and stem() is "bar"
path("/foo/bar").extension(); // yields "" and stem() is "bar"
path("/foo/.profile").extension(); // yields "" and stem() is ".profile"
path(".bar").extension(); // yields ".bar" and stem() is "bar"
path("..bar").extension(); // yields ".bar" and stem() is "..

— end example

[Note: The period is included in the return value so that it is possible to distinguish between no extension and an empty extension. — end note]

[Note: On non-POSIX operating systems, for a path p, it may not be the case that p.stem() + p.extension() == p.filename(), even though the generic format pathnames are the same. — end note]

29.11.7.4.10 Query

[[nodiscard]] bool empty() const noexcept;
Returns: true if the pathname in the generic format is empty, else 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), else 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]

```cpp
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 root_name() != base.root_name() is true or is_absolute() != base.is_absolute() is true or !has_root_directory() & base.has_root_directory() is true, returns path(). Determines the first mismatched element of *this and base as if by:

```cpp
auto [a, b] = mismatch(begin(), end(), base.begin(), base.end());
```

Then,

1. if a == end() and b == base.end(), returns path(", "); otherwise
2. let n be the number of filename elements in [b, base.end()) that are not dot or dot-dot or empty, minus the number that are dot-dot. If n<0, returns path(); otherwise
3. if n == 0 and (a == end() || a->empty()), returns path("."); otherwise
4. returns an object of class path that is default-constructed, followed by
   - application of operator/=(path("..")) n times, and then
   - application of operator/= for each element in [a, end()).

**Example:**

```cpp
assert(path("/a/d").lexically_relative("/a/b/c") == "./..//d");
assert(path("/a/b/c") .lexically_relative("/a/d") == ".//b/c");
assert(path("a/b/c") .lexically_relative("a") == "b/c");
assert(path("a/b/c") .lexically_relative("a/b/c/x/y") == ".//..");
assert(path("a/b/c") .lexically_relative("a/b/c") == "");
assert(path("a/b") .lexically_relative("c/d") == ".//a/b");
```

The above assertions will succeed. On Windows, the returned path’s directory-separator characters will be backslashes rather than slashes, but that does not affect path equality. — end example]

```cpp
path lexically_proximate(const path& base) const;
```

**Returns:** If the value of lexically_relative(base) is not an empty path, return it. Otherwise return *this.

**Note:** If symlink following semantics are desired, use the operational function proximate(). — end note

[29.11.7.5 Iterators [fs.path.itr]]

1. Path iterators iterate over the elements of the pathname in the generic format (29.11.7.1).
2. A path::iterator is a constant iterator satisfying all the requirements of a bidirectional iterator (23.3.5.5) except that, for dereferenceable iterators a and b of type path::iterator with a == b, there is no requirement that *a and *b are bound to the same object. Its value_type is path.
3. Calling any non-const member function of a path object invalidates all iterators referring to elements of that object.
4. For the elements of the pathname in the generic format, the forward traversal order is as follows:
   - The root-name element, if present.
   - The root-directory element, if present. [Note: The generic format is required to ensure lexicographical comparison works correctly. — end note]
(4.3) — Each successive `filename` element, if present.
(4.4) — An empty element, if a trailing non-root `directory-separator` is present.

The backward traversal order is the reverse of forward traversal.

iterator begin() const;

Returns: An iterator for the first present element in the traversal list above. If no elements are present, the end iterator.

iterator end() const;

Returns: The end iterator.

29.11.7.6 Inserter and extractor [fs.path.io]

```cpp
template<class charT, class traits>
friend basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const path& p);

Effects: Equivalent to `os << quoted(p.string<charT, traits>())`. [Note: The `quoted` function is described in 29.7.8. — end note]

Returns: `os`.
```

```cpp
template<class charT, class traits>
friend basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, path& p);

Effects: Equivalent to:

```cpp
    basic_string<charT, traits> tmp;
    is >> quoted(tmp);
    p = tmp;
```

Returns: `is`.
```

29.11.7.7 Non-member functions [fs.path.nonmember]

```cpp
void swap(path& lhs, path& rhs) noexcept;
```

Effects: Equivalent to `lhs.swap(rhs).

```cpp
size_t hash_value (const path& p) noexcept;
```

Returns: A hash value for the path `p`. If for two paths, `p1 == p2` then `hash_value(p1) == hash_value(p2).

```cpp
friend bool operator==(const path& lhs, const path& rhs) noexcept;
```

Returns: `!(lhs < rhs) && !(rhs < lhs).

[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]

```cpp
friend bool operator!=(const path& lhs, const path& rhs) noexcept;
```

Returns: `!(lhs == rhs).

```cpp
friend bool operator< (const path& lhs, const path& rhs) noexcept;
```

Returns: `lhs.compare(rhs) < 0`.

§ 29.11.7
friend bool operator<=(const path& lhs, const path& rhs) noexcept;

Returns: !(rhs < lhs).

friend bool operator> (const path& lhs, const path& rhs) noexcept;

Returns: rhs < lhs.

friend bool operator>=(const path& lhs, const path& rhs) noexcept;

Returns: !(lhs < rhs).

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;
    };
}

The class filesystem_error defines the type of objects thrown as exceptions to report file system errors from functions described in this subclause.

29.11.8.1 Members

Constructors are provided that store zero, one, or two paths associated with an error.

filesystem_error(const string& what_arg, error_code ec);

Ensures:
(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);

Ensures:
(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);

Ensures:
(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

29.11.9.1 Enum path::format

This enum specifies constants used to identify the format of the character sequence, with the meanings listed in Table 116.

Table 116 — 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

This enum class specifies constants used to identify file types, with the meanings listed in Table 117. The values of the constants are distinct.

Table 117 — Enum class file_type

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>The type of the file has not been determined or an error occurred while</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>found is not considered an error while determining the type of a file.</td>
</tr>
<tr>
<td></td>
<td>— 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</td>
</tr>
<tr>
<td></td>
<td>the above file_type types shall supply implementation-defined file_type</td>
</tr>
<tr>
<td></td>
<td>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`  

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 118. 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>
<thead>
<tr>
<th>Option group controlling <code>copy_file</code> 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 <code>copy</code> 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 <code>copy</code> 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 <code>copy</code> 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 119.

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 120. The bitmask constants are bitmask elements. In Table 120 perm denotes a value of type `perms` passed to `permissions`.

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 121. The constant none represents the empty bitmask; every other constant in the table represents a distinct bitmask element.

29.11.10  Class `file_status`  

```cpp
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;
```
Table 119 — Enum class `perms`

<table>
<thead>
<tr>
<th>Name</th>
<th>Value (octal)</th>
<th>POSIX macro</th>
<th>Definition or notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0</td>
<td></td>
<td>There are no permissions set for the file.</td>
</tr>
<tr>
<td>owner_read</td>
<td>0400</td>
<td>S_IRUSR</td>
<td>Read permission, owner</td>
</tr>
<tr>
<td>owner_write</td>
<td>0200</td>
<td>S_IWUSR</td>
<td>Write permission, owner</td>
</tr>
<tr>
<td>owner_exec</td>
<td>0100</td>
<td>S_IXUSR</td>
<td>Execute/search permission, owner</td>
</tr>
<tr>
<td>owner_all</td>
<td>0700</td>
<td>S_IRWXU</td>
<td>Read, write, execute/search by owner;</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>`owner_all</td>
</tr>
<tr>
<td>set_uid</td>
<td>04000</td>
<td>S_ISUID</td>
<td>Set-user-ID on execution</td>
</tr>
<tr>
<td>set_gid</td>
<td>02000</td>
<td>S_ISGID</td>
<td>Set-group-ID on execution</td>
</tr>
<tr>
<td>sticky_bit</td>
<td>01000</td>
<td>S_ISVTX</td>
<td>Operating system dependent.</td>
</tr>
<tr>
<td>mask</td>
<td>07777</td>
<td></td>
<td>`all</td>
</tr>
<tr>
<td>unknown</td>
<td>0xFFFF</td>
<td></td>
<td>The permissions are not known, such as when a <code>file_status</code> object is created without specifying the permissions</td>
</tr>
</tbody>
</table>

Table 120 — 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 <code>perm</code></td>
</tr>
<tr>
<td>add</td>
<td>permissions shall replace the file's permission bits with the bitwise OR of <code>perm</code> 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 <code>perm</code> 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>

Table 121 — 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>
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;

1 An object of type file_status stores information about the type and permissions of a file.

29.11.10.1 Constructors [fs.file.status.cons]
explicit file_status(file_type ft, perms prms = perms::unknown) noexcept;

1 Ensures: type() == ft and permissions() == prms.

29.11.10.2 Observers [fs.file.status.obs]
file_type type() const noexcept;

1 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;

2 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 [fs.file.status.mods]
void type(file_type ft) noexcept;

1 Ensures: type() == ft.

void permissions(perms prms) noexcept;

2 Ensures: permissions() == prms.

29.11.11 Class directory_entry [fs.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_symlink() const;
bool is_symlink(error_code& ec) const noexcept;
uintmax_t file_size() const;
uintmax_t file_size(error_code& ec) const noexcept;
uintmax_t hard_link_count() const;
uintmax_t hard_link_count(error_code& ec) const noexcept;
file_time_type last_write_time() const;
file_time_type last_write_time(error_code& ec) const noexcept;
file_status status() const;
file_status status(error_code& ec) const noexcept;
file_status symlink_status() const;
file_status symlink_status(error_code& ec) const noexcept;

bool operator==(const directory_entry& rhs) const noexcept;
bool operator!=(const directory_entry& rhs) const noexcept;
bool operator< (const directory_entry& rhs) const noexcept;
bool operator> (const directory_entry& rhs) const noexcept;
bool operator<=(const directory_entry& rhs) const noexcept;
bool operator>=(const directory_entry& rhs) const noexcept;

private:
    filesystem::path pathobject; // exposition only
friend class directory_iterator; // exposition only
};

A directory_entry object stores a path object and may store additional objects for file attributes such as
hard link count, status, symlink status, file size, and last write time.

Implementations should store such additional file attributes during directory iteration if their values are avail-
able and storing the values would allow the implementation to eliminate file system accesses by directory_-
entry observer functions (29.11.14). 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:
    using namespace std::filesystem;

§ 29.11.11 1414
// 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  [fs.dir.entry.cons]

```cpp
effects
explicit directory_entry(const filesystem::path& p);
directory_entry(const filesystem::path& p, error_code& ec);
effects
Constructs an object of type directory_entry, then refresh() or refresh(ec), respectively.
ensures
path() == p if no error occurs, otherwise path() == filesystem::path().
throws
As specified in 29.11.6.
```

29.11.11.2 Modifiers  [fs.dir.entry.mods]

```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.
```

```cpp
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]

```cpp
const filesystem::path& path() const noexcept;
operator const filesystem::path&() const noexcept;
returns
pathobject.
```

§ 29.11.11.3 1415
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.

bool operator!=(const directory_entry& rhs) const noexcept;

Returns: pathobject != rhs.pathobject.

bool operator< (const directory_entry& rhs) const noexcept;

Returns: pathobject < rhs.pathobject.

bool operator> (const directory_entry& rhs) const noexcept;

Returns: pathobject > rhs.pathobject.

bool operator<=(const directory_entry& rhs) const noexcept;

Returns: pathobject <= rhs.pathobject.

bool 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;
        explicit directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
directory_iterator(const path& p, error_code& ec);
directory_iterator(const path& p, directory_options options, 
    error_code& ec);
directory_iterator(const directory_iterator& rhs);
directory_iterator(directory_iterator&& rhs) noexcept;
~directory_iterator();
directory_iterator& operator=(const directory_iterator& rhs);
directory_iterator& operator=(directory_iterator&& rhs) noexcept;
const directory_entry& operator*() const;
const directory_entry* operator->() const;
directory_iterator& operator++();
increment(error_code& ec);

// other members as required by 23.3.5.2, input iterators

}  // namespace filesystem

2 directory_iterator satisfies the Cpp17InputIterator requirements (23.3.5.2).  

3 If an iterator of type directory_iterator reports an error or is advanced past the last directory element, 
that iterator shall become equal to the end iterator value. The directory_iterator default constructor 
shall create an iterator equal to the end iterator value, and this shall be the only valid iterator for the end 
condition. 

4 The end iterator is not dereferenceable. 

5 Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator. 

6 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=/.

7 Directory iteration shall not yield directory entries for the current (dot) and parent (dot-dot) directories. 

8 The order of directory entries obtained by dereferencing successive increments of a directory_iterator is 
unspecified. 

9 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] 

10 [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] 

11 [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 [fs.dir.itr.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);

2 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
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;

Effects: Constructs an object of class directory_iterator.

Ensures: *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.

Ensures: *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);
        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
// 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 [fs.rec.dir.itr.members]

recursive_directory_iterator() noexcept;

1 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);

2 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.

3 Ensures: options() == options for the signatures with a directory_options argument, otherwise options() == directory_options::none.

4 Throws: As specified in 29.11.6.

5 [Note: To iterate over the current directory, use recursive_directory_iterator(".") rather than recursive_directory_iterator(""). —end note]

6 [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);

7 Effects: Constructs an object of class recursive_directory_iterator.

8 Ensures:

(8.1) — options() == rhs.options()
— depth() == rhs.depth()
— recursion_pending() == rhs.recursion_pending()

recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;

Effects: Constructs an object of class recursive_directory_iterator.
 Ensures: 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.
 Ensures:
— options() == rhs.options()
— depth() == rhs.depth()
— recursion Pending() == rhs.recursion_pending()

Returns: *this.

recursive_directory_iterator& operator=(recursive_directory_iterator&& rhs) noexcept;

Effects: If *this and rhs are the same object, the member has no effect.
 Ensures: 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:

— 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().

— Otherwise if

recursion_pending() && is_directory((*this)->status()) &&
(!is_symlink((*this)->symlink_status()) ||
 (options() & directory_options::follow_directory_symlink) != directory_options::none)

then either directory (*this)->path() is recursively iterated into or, if

(options() & directory_options::skip_permission_denied) != directory_options::none

and an error occurs indicating that permission to access directory (*this)->path() is denied, then directory (*this)->path() is treated as an empty directory and no error is reported.

Returns: *this.

Throws: As specified in 29.11.6.
void pop();
void pop(error_code& ec);

Effects: If depth() == 0, set *this to recursive_directory_iterator(). Otherwise, cease iteration of the directory currently being iterated over, and continue iteration over the parent directory.

Ensures: Any copies of the previous value of *this are no longer required to be dereferenceable nor to be in the domain of ==.

Throws: As specified in 29.11.6.

void disable_recursion_pending();

Ensures: recursion_pending() == false.

[Note: disable_recursion_pending() is used to prevent unwanted recursion into a directory. — end note]

29.11.13.2 Non-member functions

These functions enable use of recursive_directory_iterator with range-based for statements.

recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;

Returns: iter.

recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;

Returns: recursive_directory_iterator().

29.11.14 Filesystem operation functions

Filesystem operation functions query or modify files, including directories, in external storage.

[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. — end note]

29.11.14.1 Absolute

path absolute(const path& p);
path absolute(const path& p, error_code& ec);

Effects: Composes an absolute path referencing the same file system location as p according to the operating system (29.11.2.2).

Returns: The composed path. The signature with argument ec returns path() if an error occurs.

[Note: For the returned path, rp, rp.is_absolute() is true unless an error occurs. — end note]

Throws: As specified in 29.11.6.

[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). — end note]

[Example: For POSIX-based operating systems, absolute(p) is simply current_path()/p. For Windows-based operating systems, absolute might have the same semantics as GetFullPathNameW. — end example]

29.11.14.2 Canonical

path canonical(const path& p);
path canonical(const path& p, error_code& ec);

Effects: Converts p to an absolute path that has no symbolic link, dot, or dot-dot elements in its pathname in the generic format.

Returns: A path that refers to the same file system object as absolute(p). The signature with argument ec returns path() if an error occurs.

Throws: As specified in 29.11.6.

Remarks: !exists(p) is an error.
29.11.14.3 Copy

```cpp
void copy(const path& from, const path& to);
```

**Effects:** Equivalent to `copy(from, to, copy_options::none)`.

```cpp
void copy(const path& from, const path& to, error_code& ec);
```

**Effects:** Equivalent to `copy(from, to, copy_options::none, ec)`.

```cpp
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);
```

**Requires:** At most one element from each option group (29.11.9.3) is set in `options`.

**Effects:** Before the first use of `f` and `t`:

1. If
   ```cpp
   (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)`.

2. Otherwise, if
   ```cpp
   (options & copy_options::copy_symlinks) != copy_options::none
   ```
   then auto `f = symlink_status(from)` and if needed auto `t = status(to)`.

3. Otherwise, auto `f = status(from)` and if needed auto `t = status(to)`.

Effects are then as follows:

4. If `f.type()` or `t.type()` is an implementation-defined file type (29.11.9.2), then the effects are implementation-defined.

5. Otherwise, an error is reported as specified in 29.11.6 if:
   ```cpp
   - exists(f) is false, or
   - equivalent(from, to) is true, or
   - is_other(f) || is_other(t) is true, or
   - is_directory(f) && is_regular_file(t) is true.
   ```

6. Otherwise, if `is_symlink(f)`, then:
   ```cpp
   - If (options & copy_options::skip_symlinks) != copy_options::none then return.
   - Otherwise if
     ```cpp
     !exists(t) && (options & copy_options::copy_symlinks) != copy_options::none
     ```
     then `copy_symlink(from, to)`.
   ```cpp
   - Otherwise report an error as specified in 29.11.6.
   ```

7. Otherwise, if `is_regular_file(f)`, then:
   ```cpp
   - If (options & copy_options::directories_only) != copy_options::none, then return.
   - Otherwise, if (options & copy_options::create_symlinks) != copy_options::none,
     then create a symbolic link to the source file.
   - Otherwise, if (options & copy_options::create_hard_links) != copy_options::none,
     then create a hard link to the source file.
   ```

8. Otherwise, if `is_directory(t)`, then `copy_file(from, to/from.filename(), options)`.

9. Otherwise, `copy_file(from, to, options)`.

10. Otherwise, if
    ```cpp
    is_directory(f) &&
     (options & copy_options::create_symlinks) != copy_options::none
    ```
    then report an error with an `error_code` argument equal to `make_error_code(errc::is_a_directory)`.  

§ 29.11.3
Otherwise, if
\[\text{is
directory}(f) \&\&
(\text{options} \& \text{copy\_options::recursive}) \neq \text{copy\_options::none} ||
\text{options} == \text{copy\_options::none}\]
then:

1. If \text{exists}(t) is false, then \text{create\_directory}(to, from).
2. Then, iterate over the files in from, as if by
   \[\text{for\ (const\ directory\_entry\&\ x:\ directory\_iterator(from))}\]
   \[\text{copy}(x\text{.path()}, to/x\text{.path()}.filename(),
     \text{options} \mid \text{copy\_options::in\_recursive\_copy});\]
   where \text{in\_recursive\_copy} is a bitmask element of \text{copy\_options} that is not one of the
   elements in 29.11.9.3.

Otherwise, for the signature with argument \text{ec}, \text{ec}.\text{clear}().

Otherwise, no effects.

Throws: As specified in 29.11.6.

Remarks: For the signature with argument \text{ec}, any library functions called by the implementation shall
have an \text{error\_code} argument if applicable.

\textbf{Example:} Given this directory structure:

```
/dir1
  file1
  file2
  dir2
  file3
```

Calling \text{copy}("/dir1", "/dir3") would result in:

```
/dir1
  file1
  file2
  dir2
  file3

/dir3
  file1
  file2
```

Alternatively, calling \text{copy}("/dir1", "/dir3", \text{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 \text{copy\_file}(\text{const}\ \text{path}\&\ \text{from}, \text{const}\ \text{path}\&\ \text{to});
bool \text{copy\_file}(\text{const}\ \text{path}\&\ \text{from}, \text{const}\ \text{path}\&\ \text{to}, \text{error\_code}\&\ \text{ec});

\textbf{Returns:} \text{copy\_file}(\text{from}, \text{to}, \text{copy\_options::none}) or
\text{copy\_file}(\text{from}, \text{to}, \text{copy\_options::none}, \text{ec}), respectively.

\textbf{Throws:} As specified in 29.11.6.

bool \text{copy\_file}(\text{const}\ \text{path}\&\ \text{from}, \text{const}\ \text{path}\&\ \text{to}, \text{copy\_options\ options});
bool copy_file(const path& from, const path& to, copy_options options,
error_code& ec);

Requires: 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:
(4.1.1) — is_regular_file(from) is false, or
(4.1.2) — exists(to) is true and is_regular_file(to) is false, or
(4.1.3) — exists(to) is true and equivalent(from, to) is true, or
(4.1.4) — 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:
(4.2.1) — exists(to) is false, or
(4.2.2) — (options & copy_options::overwrite_existing) != copy_options::none, or
(4.2.3) — (options & copy_options::update_existing) != copy_options::none and from is
          more recent than to, determined as if by use of the last_write_time function (29.11.14.25).
(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: \(O(n)\) where \(n\) is the number of elements of p.

29.11.14.7 Create directory

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.
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()`.

**Ensures:** `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()`.

**Ensures:**

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()`.

**Ensures:** `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);

<table>
<thead>
<tr>
<th>Returns:</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throws:</td>
<td>As specified in 29.11.6.</td>
</tr>
<tr>
<td>Remarks:</td>
<td>The current working directory is the directory, associated with the process, that is used as the starting location in pathname resolution for relative paths.</td>
</tr>
</tbody>
</table>

[Note: The current_path() name was chosen to emphasize that the returned value is a path, not just a single directory name. — end note]

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().                                                                                                                                                                                                  |
| Ensures: | 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, else 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;

| Effects: | The signature with argument ec calls ec.clear() if status_known(s).                                                                                                                                                                                      |
| Returns: | exists(s).                                                                                                                                                                                                                                             |
| Throws:  | As specified in 29.11.6.                                                                                                                                                                                                                                   |

29.11.14.14 File size

uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;

| Effects: | If exists(p) is false, an error is reported (29.11.6).                                                                                                                                                                                               |
| Returns: |                                                                                                                                                                                                                                                      |
If is_regular_file(p), the size in bytes of the file p resolves to, determined as if by the value of the POSIX stat class member st_size obtained as if by POSIX stat().

Otherwise, the result is implementation-defined.

The signature with argument ec returns static_cast<uintmax_t>(-1) if an error occurs.

**29.11.14.15 Hard link count**

\[
\text{uintmax_t hard_link_count(const path& p);} \\
\text{uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;} \\
\text{Returns: The number of hard links for p. The signature with argument ec returns static_cast\text{<uintmax_t>}{(-1)} if an error occurs.}
\]

**29.11.14.16 Is block file**

\[
\text{bool is_block_file(file_status s) noexcept;} \\
\text{bool is_block_file(const path& p);} \\
\text{bool is_block_file(const path& p, error_code& ec) noexcept;} \\
\text{Returns: \text{s.type()} == file_type::block.}
\]

**29.11.14.17 Is character file**

\[
\text{bool is_character_file(file_status s) noexcept;} \\
\text{bool is_character_file(const path& p);} \\
\text{bool is_character_file(const path& p, error_code& ec) noexcept;} \\
\text{Returns: \text{s.type()} == file_type::character.}
\]

**29.11.14.18 Is directory**

\[
\text{bool is_directory(file_status s) noexcept;} \\
\text{bool is_directory(const path& p);} \\
\text{bool is_directory(const path& p, error_code& ec) noexcept;} \\
\text{Returns: \text{s.type()} == file_type::directory.}
\]

**29.11.14.19 Is empty**

\[
\text{bool is_empty(const path& p);} \\
\text{bool is_empty(const path& p, error_code& ec);} \\
\text{Effects:}
\]

(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.
— For the signature with argument \texttt{ec}, return \texttt{false} if an error occurred.

— Otherwise, return \texttt{itr == directory_iterator()}.

— Otherwise:

— Determine \texttt{uintmax_t sz}, as if by \texttt{file_size(p)} or \texttt{file_size(p, ec)}, respectively.

— For the signature with argument \texttt{ec}, return \texttt{false} if an error occurred.

— Otherwise, return \texttt{sz == 0}.

\textit{Throws:} As specified in 29.11.6.

\textbf{29.11.14.20 Is fifo} \hfill [fs.op.is.fifo]

\begin{verbatim}
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 \texttt{ec} returns \texttt{false} if an error occurs.
3    \textit{Throws:} As specified in 29.11.6.
\end{verbatim}

\textbf{29.11.14.21 Is other} \hfill [fs.op.is.other]

\begin{verbatim}
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 \texttt{ec} returns \texttt{false} if an error occurs.
3    \textit{Throws:} As specified in 29.11.6.
\end{verbatim}

\textbf{29.11.14.22 Is regular file} \hfill [fs.op.is.regular.file]

\begin{verbatim}
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    \textit{Throws:} filesystem_error if status(p) would throw filesystem_error.

bool is_regular_file(const path& p, error_code& ec) noexcept;
4    Effects: Sets \texttt{ec} as if by status(p, ec). [Note: file_type::none, file_type::not_found and
               file_type::unknown cases set \texttt{ec} to error values. To distinguish between cases, call the status function directly. —end note]
5    Returns: is_regular_file(status(p, ec)). Returns \texttt{false} if an error occurs.
\end{verbatim}

\textbf{29.11.14.23 Is socket} \hfill [fs.op.is.socket]

\begin{verbatim}
bool is_socket(file_status s) noexcept;
1    Returns: s.type() == file_type::socket.

bool is_socket(const path& p);
bool is_socket(const path& p, error_code& ec) noexcept;
2    Returns: is_socket(status(p)) or is_socket(status(p, ec)), respectively. The signature with argument \texttt{ec} returns \texttt{false} if an error occurs.
3    \textit{Throws:} As specified in 29.11.6.
\end{verbatim}
29.11.14.24 Is symlink

bool is_symlink(file_status s) noexcept;
Returns: s.type() == file_type::symlink.

bool is_symlink(const path& p);
bool is_symlink(const path& p, error_code& ec) noexcept;
Returns: is_symlink(symlink_status(p)) or is_symlink(symlink_status(p, ec)), respectively.
The signature with argument ec returns false if an error occurs.

Throws: As specified in 29.11.6.

29.11.14.25 Last write time

file_time_type last_write_time(const path& p);
file_time_type last_write_time(const path& p, error_code& ec) noexcept;
Returns: The time of last data modification of p, determined as if by the value of the POSIX stat
class member st_mtime obtained as if by POSIX stat(). The signature with argument ec returns
file_time_type::min() if an error occurs.

Throws: As specified in 29.11.6.

void last_write_time(const path& p, file_time_type new_time);
void last_write_time(const path& p, file_time_type new_time,
error_code& ec) noexcept;
Effects: Sets the time of last data modification of the file resolved to by p to new_time, as if by POSIX
futimens().

Throws: As specified in 29.11.6.

[Note: A postcondition of last_write_time(p) == new_time is not specified since it might not hold
for file systems with coarse time granularity. — end note]

29.11.14.26 Permissions

void permissions(const path& p, perms prms, perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
void permissions(const path& p, perms prms, perm_options opts, error_code& ec);
Requires: 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().
[Note: Conceptually permissions are viewed as bits, but the actual implementation may use some other
mechanism. — end note]

Throws: As specified in 29.11.6.

29.11.14.27 Proximate

path proximate(const path& p, error_code& ec);
Returns: proximate(p, current_path(), ec).

Throws: As specified in 29.11.6.

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);
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]

**Ensures:** `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]

**Ensures:** `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:
If \( \text{old}_p \) and \( \text{new}_p \) resolve to the same existing file, no action is taken.

Otherwise, the rename may include the following effects:

1. If \( \text{new}_p \) resolves to an existing non-directory file, \( \text{new}_p \) is removed; otherwise,
2. If \( \text{new}_p \) resolves to an existing directory, \( \text{new}_p \) 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.  

---

**29.11.14.33 Resize file**  
[fs.op.resize.file]

```cpp
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**  
[fs.op.space]

```cpp
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_freesize` 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.  

---

**29.11.14.35 Status**  
[fs.op.status]

```cpp
file_status status(const path& p);
```

**Effects:** As if:

```cpp
    error_code ec;
    file_status result = status(p, ec);
    if (result.type() == file_type::none)
        throw filesystem_error(implementation-supplied-message, p, ec);
```

**Returns:** See above.

**Throws:** `filesystem_error`.  

---

**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)`.  

**Returns:**

---

1. If `ec != error_code()`:
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`.  

---

§ 29.11.14.35  1432
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).

Otherwise, if the specific error indicates that p can be resolved but the attributes cannot be determined, returns file_status(file_type::unknown).

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]

Otherwise,

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]

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]

Otherwise, if the attributes indicate a block special file, as if by POSIX S_ISBLK, returns file_status(file_type::block, prms).

Otherwise, if the attributes indicate a character special file, as if by POSIX S_ISCHR, returns file_status(file_type::character, prms).

Otherwise, if the attributes indicate a fifo or pipe file, as if by POSIX S_ISFIFO, returns file_status(file_type::fifo, prms).

Otherwise, if the attributes indicate a socket, as if by POSIX S_ISSOCK, returns file_status(file_type::socket, prms).

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.

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.

bool status_known(file_status s) noexcept;

Returns: s.type() != file_type::none.

file_status symlink_status(const path& p);

file_status symlink_status(const path& p, error_code& ec) noexcept;

Effects: Same as status(), above, except that the attributes of p are determined as if by using POSIX lstat() to obtain a POSIX struct stat.

Let prms denote the result of (m & perms::mask), where m is determined as if by converting the st_mode member of the obtained struct stat to the type perms.

Returns: Same as status(), above, except that if the attributes indicate a symbolic link, as if by POSIX S_ISLINK, 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.
path temp_directory_path(error_code& ec);

1. Let \( p \) be an unspecified directory path suitable for temporary files.

2. **Effects:** If `exists(p)` is `false` or `is_directory(p)` is `false`, an error is reported (29.11.6).

3. **Returns:** The path \( p \). The signature with argument \( ec \) returns `path()` if an error occurs.

4. **Throws:** As specified in 29.11.6.

5. *[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

[fs.op.weakly.canonical]

path weakly_canonical(const path& p);

path weakly_canonical(const path& p, error_code& ec);

1. **Returns:** \( p \) with symlinks resolved and the result normalized (29.11.7.1).

2. **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.

3. **Ensures:** The returned path is in normal form (29.11.7.1).

4. **Remarks:** Implementations should avoid unnecessary normalization such as when `canonical` has already been called on the entirety of \( p \).

5. **Throws:** As specified in 29.11.6.

### 29.12 C library files

[c.files]

#### 29.12.1 Header `<cstdio>` synopsis

[cstdio.syn]

```cpp
namespace std {
    using size_t = see 17.2.4;
    using FILE = see below;
    using fpos_t = see below;
}
```

```cpp
#define NULL see 17.2.3
#define _IOFBF see below
#define _IOLBF see below
#define _IONBF see below
#define BUFSIZ see below
#define EOF see below
#define FOPEN_MAX see below
#define FILENAME_MAX see below
#define L_tmpnam see below
#define SEEK_CUR see below
#define SEEK_END see below
#define SEEK_SET see below
#define TMP_MAX see below
#define stderr see below
#define stdin see below
#define stdout see below
```

```cpp
namespace std {
    int remove(const char* filename);
    int rename(const char* old, const char* new);
    FILE* tmpfile();
    char* tmpnam(char* s);}
```
int fclose(FILE* stream);
int fflush(FILE* stream);
FILE* fopen(const char* filename, const char* mode);
void setbuf(FILE* stream, char* buf);
int setvbuf(FILE* stream, char* buf, int mode, size_t size);
int fprintf(FILE* stream, const char* format, ...);
int fscanf(FILE* stream, const char* format, ...);
int printf(const char* format, ...);
int scanf(const char* format, ...);
int snprintf(char* s, size_t n, const char* format, ...);
int sprintf(char* s, const char* format, ...);
int sscanf(const char* s, const char* format, ...);
int vfprintf(FILE* stream, const char* format, va_list arg);
int vfscanf(FILE* stream, const char* format, va_list arg);
int vprintf(const char* format, va_list arg);
int vscanf(const char* format, va_list arg);
int vsnprintf(char* s, size_t n, const char* format, va_list arg);
int vsprintf(char* s, const char* format, va_list arg);
int vsscanf(const char* s, const char* format, va_list arg);
int fgetc(FILE* stream);
char* fgets(char* s, int n, FILE* stream);
int fputc(int c, FILE* stream);
int fputs(const char* s, FILE* stream);
int getc(FILE* stream);
int getchar();
int putc(int c, FILE* stream);
int putchar(int c);
int puts(const char* s);
int ungetc(int c, FILE* stream);
size_t fread(void* ptr, size_t size, size_t nmemb, FILE* stream);
size_t fwrite(const void* ptr, size_t size, size_t nmemb, FILE* stream);
int fgetpos(FILE* stream, fpos_t* pos);
int fseek(FILE* stream, long int offset, int whence);
int fsetpos(FILE* stream, const fpos_t* pos);
long int ftell(FILE* stream);
void rewind(FILE* stream);
void clearerr(FILE* stream);
int feof(FILE* stream);
int ferror(FILE* stream);
void perror(const char* s);
}

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 (16.5.5.9) 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

```c
#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 strtonumax(const char* nptr, char** endptr, int base);
uintmax_t strtoumax(const char* nptr, char** endptr, int base);
intmax_t wcstomax(const wchar_t* nptr, wchar_t** endptr, int base);
uintmax_t wcstoumax(const wchar_t* nptr, wchar_t** endptr, int base);

intmax_t abs(intmax_t); // optional, see below
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 `<stddef.h>` instead of `<cstdint.h>`, and
— if and only if the type `intmax_t` designates an extended integer type (6.7.1), the following function signatures are added:

```c
intmax_t abs(intmax_t);
imaxdiv_t div(intmax_t, intmax_t);
```

which shall have the same semantics as the function signatures `intmax_t imaxabs(intmax_t)` and `imaxdiv_t imaxdiv(intmax_t, intmax_t)`, respectively.

See also: ISO C 7.8
30  Regular expressions library [re]

30.1  General [re.general]

1. This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

2. The following subclauses describe a basic regular expression class template and its traits that can handle char-like (21.1) template arguments, two specializations of this class template that handle sequences of char and wchar_t, a class template that holds the result of a regular expression match, a series of algorithms that allow a character sequence to be operated upon by a regular expression, and two iterator types for enumerating regular expression matches, as summarized in Table 122.

Table 122 — Regular expressions library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</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>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  [defns.regex.collating.element]
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  [defns.regex.finite.state.machine]
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  [defns.regex.format.specifier]
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  [defns.regex.matched]
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  [defns.regex.primary.equivalence.class]
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 regular expression
a pattern that selects specific strings from a set of character strings.

30.2.7 sub-expression
a subset of a regular expression that has been marked by parenthesis.

30.3 Requirements

1 This subclause defines requirements on classes representing regular expression traits. [Note: The class template regex_traits, defined in 30.7, satisfies 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 123 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 123 — Regular expression traits class requirements

<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 123 — 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><code>v.transform_primary(F1, F2)</code></td>
<td><code>X::string_type</code></td>
<td>Returns a sort key for the character sequence designated by the iterator range <code>[F1, F2)</code> such that if the character sequence <code>[G1, G2)</code> sorts before the character sequence <code>[H1, H2)</code> when character case is not considered then <code>v.transform_primary(G1, G2) &lt; v.transform_primary(H1, H2)</code>.</td>
</tr>
<tr>
<td><code>v.lookup_collatename(F1, F2)</code></td>
<td><code>X::string_type</code></td>
<td>Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range <code>[F1, F2)</code>. Returns an empty string if the character sequence is not a valid collating element.</td>
</tr>
<tr>
<td><code>v.lookup_classname(F1, F2, b)</code></td>
<td><code>X::char_class_type</code></td>
<td>Converts the character sequence designated by the iterator range <code>[F1, F2)</code> 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 <code>b</code> 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><code>v.isctype(c, cl)</code></td>
<td><code>bool</code></td>
<td>Returns true if character <code>c</code> is a member of one of the character classes designated by <code>cl</code>, false otherwise.</td>
</tr>
<tr>
<td><code>v.value(c, I)</code></td>
<td><code>int</code></td>
<td>Returns the value represented by the digit <code>c</code> in base <code>I</code> if the character <code>c</code> is a valid digit in base <code>I</code>; otherwise returns -1. [Note: The value of <code>I</code> will only be 8, 10, or 16. — end note]</td>
</tr>
</tbody>
</table>
// 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 wregex = 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>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

template<class BiIter, class ST, class SA>
bool operator>=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& 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>
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>
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>
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>
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>
bool operator>=(const sub_match<BiIter>& lhs,
               const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template<class BiIter>
bool operator==(const typename iterator_traits<BiIter>::value_type* lhs,
                const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator!=(const typename iterator_traits<BiIter>::value_type* lhs,
                const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator<(const typename iterator_traits<BiIter>::value_type* lhs,
               const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type* lhs,
               const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator<=(const typename iterator_traits<BiIter>::value_type* lhs,
               const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator>=(const typename iterator_traits<BiIter>::value_type* lhs,
               const sub_match<BiIter>& rhs);

§ 30.4 1442
template<class BiIter>
bool operator==(const typename iterator_traits<BiIter>::value_type& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator!=(const typename iterator_traits<BiIter>::value_type& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator<(const typename iterator_traits<BiIter>::value_type& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator<=(const typename iterator_traits<BiIter>::value_type& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool operator>=(const typename iterator_traits<BiIter>::value_type& lhs, const sub_match<BiIter>& rhs);

template<class BiIter>
bool 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>
bool 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>
bool 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 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>;

// 30.10.7, match_results swap
template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1, match_results<BidirectionalIterator, Allocator>& m2);
// 30.11.2, function template regex_match  
template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,  
    match_results<BidirectionalIterator, Allocator>& m,  
    const basic_regex<charT, traits>& e,  
    regex_constants::match_flag_type flags = regex_constants::match_default);  

// 30.11.3, function template regex_search  
template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,  
    match_results<BidirectionalIterator, Allocator>& m,  
    const basic_regex<charT, traits>& e,  
    regex_constants::match_flag_type flags = regex_constants::match_default);  

§ 30.4
template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s, Allocator& a, const basic_regex<charT, traits>& re, regex_constants::match_flag_type flags = regex_constants::match_default) = delete;

// 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, 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 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 charT* fmt, regex_constants::match_flag_type flags = regex_constants::match_default);

template<class OutputIterator, class BidirectionalIterator, class traits, class charT, class ST, class SA>
basic_string<charT, ST, SA> regex_replace(const charT* s, const basic_regex<charT, traits>& e, const charT* fmt, regex_constants::match_flag_type flags = regex_constants::match_default);

§ 30.4 1445
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;
}

1 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 124. 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;
    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 § 30.5.2
Table 124 — syntax_option_type effects

<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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>when a regular expression is matched against a character container sequence,</td>
</tr>
<tr>
<td></td>
<td>no sub-expression matches shall be stored in the supplied match_results</td>
</tr>
<tr>
<td></td>
<td>object.</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies that the regular expression engine should pay more attention to the</td>
</tr>
<tr>
<td></td>
<td>speed with which regular expressions are matched, and less to the speed with</td>
</tr>
<tr>
<td></td>
<td>which regular expression objects are constructed. Otherwise it has no</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>be that used by ECMAScript in ECMA-262, as modified in 30.13.</td>
</tr>
<tr>
<td></td>
<td>See also: ECMA-262 15.10</td>
</tr>
<tr>
<td>basic</td>
<td>Specifies that the grammar recognized by the regular expression engine shall</td>
</tr>
<tr>
<td></td>
<td>be that used by basic regular expressions in POSIX.</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>that used by extended regular expressions in POSIX.</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>end of a line, if the ECMAScript engine is selected.</td>
</tr>
</tbody>
</table>

Table 125 — regex_constants::match_flag_type effects when obtaining a match against a character container sequence (first, last).

<table>
<thead>
<tr>
<th>Element</th>
<th>Effect(s) if set</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_not_bol</td>
<td>The first character in the sequence (first, last) shall be treated as though</td>
</tr>
<tr>
<td></td>
<td>it is not at the beginning of a line, so the character ^ in the regular</td>
</tr>
<tr>
<td></td>
<td>expression shall not match (first, first).</td>
</tr>
<tr>
<td>match_not_eol</td>
<td>The last character in the sequence (first, last) shall be treated as though</td>
</tr>
<tr>
<td></td>
<td>it is not at the end of a line, so the character &quot;$&quot; in the regular expression</td>
</tr>
<tr>
<td></td>
<td>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>
</tbody>
</table>
Table 125 — 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_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_cctype = 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_space = unspecified;
inline constexpr error_type error_badrepeat = 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 126:

<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_cctype</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>
<tr>
<td>error_brace</td>
<td>The expression contained mismatched { and }</td>
</tr>
<tr>
<td>error_badbrace</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>
</tbody>
</table>
Table 126 — error_type values in the C locale (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_badrepeat</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

```cpp
class regex_error : public runtime_error {
    public:
        explicit regex_error(regex_constants::error_type ecode);
        regex_constants::error_type code() const;
    };
```

1 The class regex_error defines the type of objects thrown as exceptions to report errors from the regular expression library.

```cpp
regex_error(regex_constants::error_type ecode);
```

2 Effects: Constructs an object of class regex_error.

```cpp
regex_constants::error_type code() const;
```

3 Ensures: ecode == code().

```cpp
regex_constants::error_type code() const;
```

4 Returns: The error code that was passed to the constructor.

30.7 Class template regex_traits

```cpp
namespace std {
    template<class charT>
    struct regex_traits {
        using char_type       = charT;
        using string_type     = basic_string<char_type>;
        using locale_type     = locale;
        using char_class_type = bitmask_type;

        regex_traits();
        static size_t length(const char_type* p);
        charT translate(charT c) const;
        charT translate_nocase(charT c) const;
        template<class ForwardIterator>
            string_type transform(ForwardIterator first, ForwardIterator last) const;
        template<class ForwardIterator>
            string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
        template<class ForwardIterator>
            string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
        template<class ForwardIterator>
            char_class_type lookup_classname(ForwardIterator first, ForwardIterator last, bool icase = false) const;
        bool isctype(charT c, char_class_type f) const;
        int value(charT ch, int radix) const;
        locale_type imbue(locale_type l);
        locale_type getloc() const;
    };
}
```

1 The specializations regex_traits<char> and regex_traits<wchar_t> shall be valid and shall satisfy the requirements for a regular expression traits class (30.3).

§ 30.7
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.

static size_t length(const char_type* p);

Returns: char_traits<charT>::length(p).

charT translate(charT c) const;

Returns: c.

charT translate_nocase(charT c) const;

Returns: use_facet<ctype<charT>>(getloc()).tolower(c).

template<class ForwardIterator>
string_type transform(ForwardIterator first, ForwardIterator last) const;

Effects: As if by:

    string_type str(first, last);
    return use_facet<collate<charT>>(getloc()).transform(str.data(), str.data() + str.length());

template<class ForwardIterator>
string_type transform_primary(ForwardIterator first, ForwardIterator last) const;

Effects: If

    typeid(use_facet<collate<charT>>) == typeid(collate_byname<charT>)

and the form of the sort key returned by collate_byname<charT>::transform(first, last) is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

template<class ForwardIterator>
string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;

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.

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.\(^{329}\) 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 127 shall be recognized. For regex_traits<wchar_t>, at least the wide character names in Table 127 shall be recognized.

bool isctype(charT c, char_class_type f) const;

Effects: Determines if the character c is a member of the character classification represented by f.

Returns: Given the following function declaration:

    // for exposition only
    template<class C>
    ctype_base::mask convert(typename regex_traits<C>::char_class_type f);

that returns a value in which each ctype_base::mask value corresponding to a value in f named in Table 127 is set, then the result is determined as if by:

\(^{329}\) For example, if the parameter icase is true then [[:lower:]] is the same as [[:alpha:]].
ctype_base::mask m = convert<charT>(f);
const ctype<charT>& ct = use_facet<ctype<charT>>(getloc());
if (ct.is(m, c)) {
  return true;
} else if (c == ct.widen('_')) {
  charT w[1] = { ct.widen('w') };
  char_class_type x = lookup_classname(w, w+1);
  return (f&x) == x;
} else {
  return false;
}

[Example:
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());
ctype_base::mask m = convert<char>(f); // m == ctype_base::digit|ctype_base::upper
— end example] [Example:
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
f.isctype('_', f); // returns true
f.isctype(' ', f); // returns false
— end example]

int value(charT ch, int radix) const;
13 Requires: The value of radix shall be 8, 10, or 16.
14 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);
15 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]
16 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.
17 Ensures: getloc() == loc.

locale_type getloc() const;
18 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>
<thead>
<tr>
<th>Narrow character name</th>
<th>Wide character name</th>
<th>Corresponding <code>ctype_base::mask</code> value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;alnum&quot;</td>
<td>L&quot;alnum&quot;</td>
<td><code>ctype_base::alnum</code></td>
</tr>
<tr>
<td>&quot;alpha&quot;</td>
<td>L&quot;alpha&quot;</td>
<td><code>ctype_base::alpha</code></td>
</tr>
<tr>
<td>&quot;blank&quot;</td>
<td>L&quot;blank&quot;</td>
<td><code>ctype_base::blank</code></td>
</tr>
<tr>
<td>&quot;cntrl&quot;</td>
<td>L&quot;cntrl&quot;</td>
<td><code>ctype_base::cntrl</code></td>
</tr>
<tr>
<td>&quot;digit&quot;</td>
<td>L&quot;digit&quot;</td>
<td><code>ctype_base::digit</code></td>
</tr>
<tr>
<td>&quot;d&quot;</td>
<td>L&quot;d&quot;</td>
<td><code>ctype_base::digit</code></td>
</tr>
<tr>
<td>&quot;graph&quot;</td>
<td>L&quot;graph&quot;</td>
<td><code>ctype_base::graph</code></td>
</tr>
<tr>
<td>&quot;lower&quot;</td>
<td>L&quot;lower&quot;</td>
<td><code>ctype_base::lower</code></td>
</tr>
<tr>
<td>&quot;print&quot;</td>
<td>L&quot;print&quot;</td>
<td><code>ctype_base::print</code></td>
</tr>
<tr>
<td>&quot;punct&quot;</td>
<td>L&quot;punct&quot;</td>
<td><code>ctype_base::punct</code></td>
</tr>
<tr>
<td>&quot;space&quot;</td>
<td>L&quot;space&quot;</td>
<td><code>ctype_base::space</code></td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>L&quot;s&quot;</td>
<td><code>ctype_base::space</code></td>
</tr>
<tr>
<td>&quot;upper&quot;</td>
<td>L&quot;upper&quot;</td>
<td><code>ctype_base::upper</code></td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>L&quot;w&quot;</td>
<td><code>ctype_base::alnum</code></td>
</tr>
<tr>
<td>&quot;xdigit&quot;</td>
<td>L&quot;xdigit&quot;</td>
<td><code>ctype_base::xdigit</code></td>
</tr>
</tbody>
</table>

```cpp
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 = regex_constants::ECMAScript);
        basic_regex(const charT* p, size_t len, flag_type = regex_constants::ECMAScript);
        basic_regex(const basic_regex&);
        basic_regex(basic_regex&&) noexcept;
        template<class ST, class SA>
        explicit basic_regex(const basic_string<charT, ST, SA>& p,
                              flag_type = regex_constants::ECMAScript);
        template<class ForwardIterator>
        basic_regex(ForwardIterator first, ForwardIterator last,
                      flag_type = regex_constants::ECMAScript);
        basic_regex(initializer_list<charT>, flag_type = regex_constants::ECMAScript);

    ~basic_regex();
}
```
basic_regex& operator=(const basic_regex&);
basic_regex& operator=(basic_regex&) noexcept;
basic_regex& operator=(const charT* ptr);
basic_regex& operator=(initializer_list<charT> il);
template<class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>& p);

// 30.8.2, assign
basic_regex& assign(const basic_regex& that);
basic_regex& assign(basic_regex&& that) noexcept;
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);
basic_regex& assign(const charT* p, size_t len, flag_type f);  
template<class string_traits, class A>
    basic_regex& assign(const basic_string<charT, string_traits, A>& 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 = 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&);
}

30.8.1 Constructors
[re.regex.construct]

basic_regex();

Effects: Constructs an object of class basic_regex that does not match any character sequence.

explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);  

Requires: p shall not be a null pointer.

Throws: regex_error if p is not a valid regular expression.

Effects: Constructs an object of class basic_regex; the object’s internal finite state machine is
constructed from the regular expression contained in the array of charT of length char_traits<charT>::
length(p) whose first element is designated by p, and interpreted according to the flags f.

Ensures: flags() returns f. mark_count() returns the number of marked sub-expressions within the
expression.

basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);  

Requires: p shall not be a null pointer.

Throws: regex_error if p is not a valid regular expression.

Effects: Constructs an object of class basic_regex; 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 to the flags specified in f.
Ensures: \texttt{flags()} returns \( f \). \texttt{mark_count()} returns the number of marked sub-expressions within the expression.

\texttt{basic\_regex(const basic\_regex\& e);}  
\textbf{Effects}: Constructs an object of class \texttt{basic\_regex} as a copy of the object \( e \).

\texttt{basic\_regex(basic\_regex\&\& e) noexcept;}  
\textbf{Effects}: Move constructs an object of class \texttt{basic\_regex} from \( e \).

\texttt{template<class ST, class SA>}  
\texttt{explicit basic\_regex(const basic\_string&lt;charT, ST, SA&gt;\& s, \flag\_type f = regex\_constants::ECMAScript);}  
\textbf{Throws}: \texttt{regex\_error} if \( s \) is not a valid regular expression.

\texttt{Effects}: Constructs an object of class \texttt{basic\_regex}: 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 \( f \).

\texttt{Ensures: flags()} returns \( f \). \texttt{mark\_count()} returns the number of marked sub-expressions within the expression.

\texttt{template<class ForwardIterator>}  
\texttt{basic\_regex(ForwardIterator first, ForwardIterator last, \flag\_type f = regex\_constants::ECMAScript);}  
\textbf{Throws}: \texttt{regex\_error} if the sequence \([\texttt{first}, \texttt{last})\) is not a valid regular expression.

\texttt{Effects}: Constructs an object of class \texttt{basic\_regex}: the object’s internal finite state machine is constructed from the regular expression contained in the sequence of characters \([\texttt{first}, \texttt{last})\), and interpreted according to the flags specified in \( f \).

\texttt{Ensures: flags()} returns \( f \). \texttt{mark\_count()} returns the number of marked sub-expressions within the expression.

\texttt{basic\_regex(initializer\_list&lt;charT\& \ il, \flag\_type f = regex\_constants::ECMAScript);}  
\textbf{Effects}: Same as \texttt{basic\_regex(il.begin(), il.end(), \textit{f})}.

\texttt{template<class ST, class SA>}  
\texttt{basic\_regex\& operator=(const basic\_regex\& e);}  
\textbf{Effects}: Copies \( e \) into \(*\texttt{this}\) and returns \(*\texttt{this}\).

\texttt{basic\_regex\& operator=(basic\_regex\&\& e) noexcept;}  
\textbf{Effects}: Move assigns from \( e \) into \(*\texttt{this}\) and returns \(*\texttt{this}\).

\texttt{basic\_regex\& operator=(const charT\* ptr);}  
\textbf{Requires}: \( \texttt{ptr} \) shall not be a null pointer.

\texttt{basic\_regex\& operator=(initializer\_list&lt;charT\& \ il); l)}  
\textbf{Effects}: Returns \texttt{assign(ptr)}.

\texttt{template<class ST, class SA>}  
\texttt{basic\_regex\& operator=(const basic\_string&lt;charT, ST, SA\& \ p);}  
\textbf{Effects}: Returns \texttt{assign(il.begin(), il.end())}.
basic_regex& assign(const basic_regex& that);

*Effects:* Equivalent to: return *this = that;

basic_regex& assign(basic_regex&& that) noexcept;

*Effects:* Equivalent to: return *this = std::move(that);

basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);

*Returns:* assign(string_type(ptr), f).

basic_regex& assign(const charT* ptr, size_t len, flag_type f = regex_constants::ECMAScript);

*Returns:* assign(string_type(ptr, len), f).

template<class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
    flag_type f = regex_constants::ECMAScript);

*Throws:* regex_error if s is not a valid regular expression.

*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.

*Ensures:* If no exception is thrown, flags() returns f and mark_count() returns the number of marked
sub-expressions within the expression.

template<class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
    flag_type f = regex_constants::ECMAScript);

*Requires:* InputIterator shall satisfy the Cpp17InputIterator requirements (23.3.5.2).

*Returns:* assign(string_type(first, last), f).

basic_regex& assign(initializer_list<charT> il,
    flag_type f = regex_constants::ECMAScript);

*Effects:* Same as assign(il.begin(), il.end(), f).

*Returns:* *this.

### 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.
Ensures: *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

```cpp
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.

```cpp
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

```cpp
constexpr sub_match();
```

Effects: Value-initializes the pair base class subobject and the member matched.

```cpp
difference_type length() const;
```

Returns: matched ? distance(first, second) : 0.

```cpp
operator string_type() const;
```

Returns: matched ? string_type(first, second) : string_type().

```cpp
string_type str() const;
```

Returns: matched ? string_type(first, second) : string_type().

```cpp
int compare(const sub_match& s) const;
```

Returns: str().compare(s.str()).

```cpp
int compare(const string_type& s) const;
```

Returns: str().compare(s).

```cpp
int compare(const value_type* s) const;
```

Returns: str().compare(s).
30.9.2 Non-member operators

```cpp
template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) == 0.
```

```cpp
template<class BiIter>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) != 0.
```

```cpp
template<class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) < 0.
```

```cpp
template<class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) > 0.
```

```cpp
template<class BiIter>
bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) <= 0.
```

```cpp
template<class BiIter>
bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) >= 0.
```

```cpp
template<class BiIter, class ST, class SA>
bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(lhs.data(), lhs.size())) == 0.
```

```cpp
template<class BiIter, class ST, class SA>
bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

Returns: !(lhs == rhs).
```

```cpp
template<class BiIter, class ST, class SA>
bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(lhs.data(), lhs.size())) > 0.
```

```cpp
template<class BiIter, class ST, class SA>
bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

Returns: rhs < lhs.
```

```cpp
template<class BiIter, class ST, class SA>
bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs, const sub_match<BiIter>& rhs);

Returns: !(rhs < lhs).
```
template<class BiIter, class ST, class SA>
bool operator>=(
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);

Returns: !(lhs < rhs).

template<class BiIter, class ST, class SA>
bool operator==(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

Returns: rhs.compare(lhs) == 0.

template<class BiIter, class ST, class SA>
bool operator!=(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

Returns: !(lhs == rhs).

template<class BiIter, class ST, class SA>
bool operator<(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

Returns: rhs.compare(lhs) > 0.
template<class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type* lhs,
const sub_match<BiIter>& rhs);

Returns: rhs < lhs.

template<class BiIter>
bool operator<=(const typename iterator_traits<BiIter>::value_type* lhs,
const sub_match<BiIter>& rhs);

Returns: !(rhs < lhs).

template<class BiIter>
bool operator>=(const typename iterator_traits<BiIter>::value_type* lhs,
const sub_match<BiIter>& rhs);

Returns: !(lhs < rhs).

template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
const typename iterator_traits<BiIter>::value_type* rhs);

Returns: lhs.compare(rhs) == 0.

template<class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
const typename iterator_traits<BiIter>::value_type* rhs);

Returns: !(lhs == rhs).

template<class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
const typename iterator_traits<BiIter>::value_type& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) < 0.

§ 30.9.2 1459
template<class BiIter>
  bool operator>(const typename iterator_traits<BiIter>::value_type& lhs,
              const sub_match<BiIter>& rhs);
Returns: rhs < lhs.

template<class BiIter>
  bool operator<=(const typename iterator_traits<BiIter>::value_type& lhs,
                 const sub_match<BiIter>& rhs);
Returns: !(rhs < lhs).

template<class BiIter>
  bool operator>=(const typename iterator_traits<BiIter>::value_type& lhs,
                 const sub_match<BiIter>& rhs);
Returns: !(lhs < rhs).

template<class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
Returns: lhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) == 0.

template<class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
Returns: !(lhs == rhs).

template<class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
Returns: rhs < lhs.

template<class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
Returns: !(lhs < rhs).

template<class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                 const typename iterator_traits<BiIter>::value_type& rhs);
Returns: !(lhs < rhs).

template<class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);
Returns: os << m.str().

30.10 Class template match_results

1 Class template match_results denotes a collection of character sequences representing the result of a regular
expression match. Storage for the collection is allocated and freed as necessary by the member functions of
class template match_results.

2 The class template match_results satisfies 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 `sub_match` object stored at index 0 represents sub-expression 0, i.e., the whole match. In this case the `sub_match` member `matched` is always `true`. The `sub_match` object stored at index `n` denotes what matched the marked sub-expression `n` within the matched expression. If the sub-expression `n` participated in a regular expression match then the `sub_match` member `matched` evaluates to `true`, and members `first` and `second` denote the range of characters `[first, second)` which formed that match. Otherwise `matched` is `false`, and members `first` and `second` point to the end of the sequence that was searched. [Note: The `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 {
        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;
        [[nodiscard]] 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 1461
```
template<OutputIter out, const char_type* fmt_first, const char_type* fmt_last, regex_constants::match_flag_type flags = regex_constants::format_default> const;

template<class OutputIter, class ST, class SA>
OutputIter
format(OutputIter out, const basic_string<char_type, ST, SA>& fmt, regex_constants::match_flag_type flags = regex_constants::format_default) const;

template<class ST, class SA>
basic_string<char_type, ST, SA>
format(const basic_string<char_type, ST, SA>& fmt, regex_constants::match_flag_type flags = regex_constants::format_default) const;

string_type
format(const char_type* fmt, regex_constants::match_flag_type flags = regex_constants::format_default) const;

allocator_type get_allocator() const;

void swap(match_results& that);

30.10.1 Constructors

explicit match_results(const Allocator& a);
Effects: Constructs an object of class match_results.
Ensures: ready() returns false. size() returns 0.

match_results(const match_results& m);
Effects: Constructs an object of class match_results, as a copy of m.

match_results(match_results&& m) noexcept;
Effects: Move constructs an object of class match_results from m satisfying the same postconditions as Table 128. Additionally, the stored Allocator value is move constructed from m.get_allocator().
Throws: Nothing.

match_results& operator=(const match_results& m);
Effects: Assigns m to *this. The postconditions of this function are indicated in Table 128.

match_results& operator=(match_results&& m);
Effects: Move-assigns m to *this. The postconditions of this function are indicated in Table 128.

Table 128 — 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

bool ready() const;

Returns: true if *this has a fully established result state, otherwise false.

30.10.3 Size

size_type size() const;

Returns: One plus the number of marked sub-expressions in the regular expression that was matched if *this represents the result of a successful match. Otherwise returns 0. [Note: 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]

size_type max_size() const;

Returns: The maximum number of sub_match elements that can be stored in *this.

[[nodiscard]] bool empty() const;

Returns: size() == 0.

30.10.4 Element access

difference_type length(size_type sub = 0) const;

Requires: ready() == true.

Returns: (*this)[sub].length().

difference_type position(size_type sub = 0) const;

Requires: ready() == true.

Returns: The distance from the start of the target sequence to (*this)[sub].first.

string_type str(size_type sub = 0) const;

Requires: ready() == true.

Returns: string_type((*this)[sub]).

const_reference operator[](size_type n) const;

Requires: ready() == true.

Returns: A reference to the sub_match object representing the character sequence that matched marked sub-expression n. If n == 0 then returns a reference to a sub_match object representing the character sequence that matched the whole regular expression. If n >= size() then returns a sub_match object representing an unmatched sub-expression.

const_reference prefix() const;

Requires: ready() == true.

Returns: A reference to the sub_match object representing the character sequence from the start of the string being matched/searched to the start of the match found.

const_reference suffix() const;

Requires: ready() == true.

Returns: A reference to the sub_match object representing the character sequence from the end of the match found to the end of the string being matched/searched.

const_iterator begin() const;
const_iterator cbegin() const;

Returns: A starting iterator that enumerates over all the sub-expressions stored in *this.

const_iterator end() const;
const_iterator cend() const;

Returns: A terminating iterator that enumerates over all the sub-expressions stored in *this.

### 30.10.5 Formatting

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;

Requires: ready() == true and OutputIter shall satisfy the requirements for a Cpp17OutputIterator (23.3.5.3).

Effects: Copies the character sequence [fmt_first, fmt_last) to OutputIter out. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within *this to which it refers. The bitmasks specified in flags determine which format specifiers and escape sequences are recognized.

Returns: out.

template<class OutputIter, class ST, class SA>
OutputIter format(
    OutputIter out,
    const basic_string<char_type, ST, SA>& fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;

Effects: Equivalent to:

\[
\text{return format(out, fmt.data(), fmt.data() + fmt.size(), flags);}
\]

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;

Requires: ready() == true.

Effects: Constructs an empty string result of type basic_string<char_type, ST, SA> and calls:

\[
\text{format(back_inserter(result), fmt, flags);}\]

Returns: result.

string_type format(
    const char_type* fmt,
    regex_constants::match_flag_type flags = regex_constants::format_default) const;

Requires: ready() == true.

Effects: Constructs an empty string result of type string_type and calls:

\[
\text{format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags);}\]

Returns: result.

### 30.10.6 Allocator

allocator_type get_allocator() const;

Returns: A copy of the Allocator that was passed to the object’s constructor or, if that allocator has been replaced, a copy of the most recent replacement.

### 30.10.7 Swap

void swap(match_results& that);

Effects: Swaps the contents of the two sequences.

Ensures: *this contains the sequence of matched sub-expressions that were in that, that contains the sequence of matched sub-expressions that were in *this.

Complexity: Constant time.
template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
match_results<BidirectionalIterator, Allocator>& m2);

Effects: As if by m1.swap(m2).

30.10.8 Non-member functions

template<class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
const match_results<BidirectionalIterator, Allocator>& m2);

Returns: true if neither match result is ready, false if one match result is ready and the other is not. If both match results are ready, returns true only if:

1. m1.empty() && m2.empty(), or
2. !m1.empty() && !m2.empty(), and the following conditions are satisfied:

(a) m1.prefix() == m2.prefix(),
(b) m1.size() == m2.size() && equal(m1.begin(), m1.end(), m2.begin()), and
(c) m1.suffix() == m2.suffix().

[Note: The algorithm equal is defined in Clause 25. — end note]

template<class BidirectionalIterator, class Allocator>
bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
const match_results<BidirectionalIterator, Allocator>& m2);

Returns: !(m1 == m2).

30.11 Regular expression algorithms

30.11.1 Exceptions

The algorithms described in this subclause 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

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);

Requires: The type BidirectionalIterator shall satisfy 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:

```cpp
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
```

—end example]

Ensures: 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 129.
Table 129 — Effects of regex_match algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>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>
<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>

```cpp
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)`.

```cpp
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)`.

```cpp
template<class ST, class SA, 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)`.

```cpp
template<class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

**Returns:** `regex_match(str, str + char_traits<charT>::length(str), e, flags)`.

```cpp
template<class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

**Returns:** `regex_match(s.begin(), s.end(), e, flags)`.

§ 30.11.2 1466
30.11.3 regex_search

```cpp
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);
```

1 **Requires**: Type `BidirectionalIterator` shall satisfy the `Cpp17BidirectionalIterator` requirements (23.3.5.5).

2 **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.

3 **Ensures**: `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 130.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>m[0].first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>m.prefix().first != m.prefix().second</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>m[0].second</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>last</td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td>m.suffix().first != m.suffix().second</td>
</tr>
<tr>
<td>m[0].first</td>
<td>The start of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].second</td>
<td>The end of the sequence of characters that matched the regular expression</td>
</tr>
<tr>
<td>m[0].matched</td>
<td>true</td>
</tr>
<tr>
<td>m[n].first</td>
<td>For all integers <code>0 &lt; n &lt; m.size()</code>, the start of the sequence that matched sub-expression n. Alternatively, if sub-expression n did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers <code>0 &lt; n &lt; m.size()</code>, 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 <code>0 &lt; n &lt; m.size()</code>, true if sub-expression n participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

```cpp
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)`.

```cpp
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)`.
template<class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
     const basic_regex<charT, traits>& e,
     regex_constants::match_flag_type flags = regex_constants::match_default);

Effects: Behaves “as if” by constructing an object what of type match_results<BidirectionalIterator> and returning regex_search(first, last, what, e, flags).

template<class charT, class traits>
bool regex_search(const charT* str,
     const basic_regex<charT, traits>& e,
     regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_search(str, str + char_traits<charT>::length(str), e, flags).

template<class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
     const basic_regex<charT, traits>& e,
     regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_search(s.begin(), s.end(), e, flags).

30.11.4 regex_replace [re.alg.replace]

template<class OutputIterator, class BidirectionalIterator, 
     class traits, class charT, class ST, class SA>
OutputIterator
regex_replace(OutputIterator out,
     BidirectionalIterator first, BidirectionalIterator last,
     const basic_regex<charT, traits>& e,
     const basic_string<charT, ST, SA>& fmt,
     regex_constants::match_flag_type flags = regex_constants::match_default);

template<class OutputIterator, class BidirectionalIterator, class traits, class charT>
OutputIterator
regex_replace(OutputIterator out,
     BidirectionalIterator first, BidirectionalIterator last,
     const basic_regex<charT, traits>& e,
     const charT* fmt,
     regex_constants::match_flag_type flags = regex_constants::match_default);

Effects: Constructs a regex_iterator object i as if by
regex_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags)
and uses i to enumerate through all of the matches m of type match_results<BidirectionalIterator>
that occur within the sequence [first, last). If no such matches are found and !(flags & regex_constants::format_no_copy), then calls
out = copy(first, last, out)
If any matches are found then, for each such match:

(1.1) If !(flags & regex_constants::format_no_copy), calls
     out = copy(m.prefix().first, m.prefix().second, out)

(1.2) Then calls
     out = m.format(out, fmt, flags)
for the first form of the function and
     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_first_only) is nonzero, then only the first match found is replaced.

Returns: out.
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.

```cpp
namespace std {
    template<class BidirectionalIterator,
             class charT = typename iterator_traits<BidirectionalIterator>::value_type,
             class traits = regex_traits<charT>>
    class regex_iterator {
        public:
            using regex_type = basic_regex<charT, traits>;
            using iterator_category = forward_iterator_tag;
            using value_type = match_results<BidirectionalIterator>;
            using difference_type = ptrdiff_t;
            using pointer = const value_type*;
            using reference = const value_type&;

```
regex_iterator();
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
 const regex_type& re,
 regex_constants::match_flag_type m = regex_constants::match_default);
regex_iterator(BidirectionalIterator, BidirectionalIterator,
 const regex_type&,
 regex_constants::match_flag_type = regex_constants::match_default) = delete;
regex_iterator(const regex_iterator&);
regex_iterator& operator=(const regex_iterator&);
bool operator==(const regex_iterator&) const;
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
 regex_constants::match_flag_type flags; // exposition only
 match_results<BidirectionalIterator> match; // exposition only
};

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 [re.regexiter.cnstr]
regex_iterator();

Effects: Constructs an end-of-sequence iterator.

regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
 const regex_type& re,
 regex_constants::match_flag_type m = regex_constants::match_default);

Effects: Initializes begin and end to a and b, respectively, sets pregex to addressof(re), sets flags
to m, then calls regex_search(begin, end, match, *pregex, flags). If this call returns false the
constructor sets *this to the end-of-sequence iterator.

30.12.1.2 Comparisons [re.regexiter.comp]

bool operator==(const regex_iterator& right) const;

Returns: true if *this and right are both end-of-sequence iterators or if the following conditions all
hold:

(1.1) — begin == right.begin,
(1.2) — end == right.end,
(1.3) — pregex == right.pregex,
(1.4) — flags == right.flags, and
(1.5) — match[0] == right.match[0];
otherwise false.

bool operator!=(const regex_iterator& right) const;

Returns: !(this == right).
30.12.1.3 Indirection

const value_type& operator*() const;

Returns: match.

const value_type* operator->() const;

Returns: &match.

30.12.1.4 Increment

regex_iterator& operator++();

Effects: Constructs a local variable start of type BidirectionalIterator and initializes it with the value of match[0].second.

If the iterator holds a zero-length match and start == end the operator sets *this to the end-of-sequence iterator and returns *this.

Otherwise, if the iterator holds a zero-length match, the operator calls:

regex_search(start, end, match, *pregex,
flags | regex_constants::match_not_null | regex_constants::match_continuous)

If the call returns true the operator returns *this. Otherwise the operator increments start and continues as if the most recent match was not a zero-length match.

If the most recent match was not a zero-length match, the operator sets flags to flags | regex_constants::match_prev_avail and calls regex_search(start, end, match, *pregex, flags). If the call returns false the iterator sets *this to the end-of-sequence iterator. The iterator then returns *this.

In all cases in which the call to regex_search returns true, match.prefix().first shall be equal to the previous value of match[0].second, and for each index i in the half-open range [0, match.size()) for which match[i].matched is true, match.position(i) shall return distance(begin, match[i].first).

[Note: This means that match.position(i) gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to regex_search. — end note]

It is unspecified how the implementation makes these adjustments.

[Note: This means that a compiler may call an implementation-specific search function, in which case a program-defined specialization of regex_search will not be called. — end note]

regex_iterator operator++(int);

Effects: As if by:

regex_iterator tmp = *this;
++(*this);
return tmp;

30.12.2 Class template regex_token_iterator

The class template regex_token_iterator is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a sub_match class template instance that represents what matched a particular sub-expression within the regular expression.

When class regex_token_iterator is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

After it is constructed, the iterator finds and stores a value regex_iterator<BidirectionalIterator> position and sets the internal count N to zero. It also maintains a sequence subs which contains a list of the sub-expressions which will be enumerated. Every time operator++ is used the count N is incremented; if N exceeds or equals subs.size(), then the iterator increments member position and sets count N to zero.
If the end of sequence is reached (position is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const sub_match<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const sub_match<BidirectionalIterator>* is returned.

It is impossible to store things into 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.

namespace std {
   template<class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT>>
   class regex_token_iterator {
      public:
         using regex_type = basic_regex<charT, traits>;
         using iterator_category = forward_iterator_tag;
         using value_type = sub_match<BidirectionalIterator>;
         using difference_type = ptrdiff_t;
         using pointer = const value_type*;
         using reference = const value_type&;

         regex_token_iterator();
         regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                              const regex_type& re,
                              int submatch = 0,
                              regex_constants::match_flag_type m =
                              regex_constants::match_default);
         regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                              const regex_type& re,
                              const vector<int>& submatches,
                              regex_constants::match_flag_type m =
                              regex_constants::match_default);
         regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                              const regex_type& re,
                              initializer_list<int> submatches,
                              regex_constants::match_flag_type m =
                              regex_constants::match_default);
         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);
         regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                              const regex_type& re,
                              int submatch = 0,
                              regex_constants::match_flag_type m =
                              regex_constants::match_default) = delete;
         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;

§ 30.12.2
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].`
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

```cpp
bool operator==(const regex_token_iterator& right) const;
1
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.
```

```cpp
bool operator!=(const regex_token_iterator& right) const;
2
Returns: !(`*this` `==` `right`).
```

### 30.12.2.3 Indirection

```cpp
const value_type& operator*() const;
1
Returns: `result`.
```

```cpp
const value_type* operator->() const;
2
Returns: `result`.
```

### 30.12.2.4 Increment

```cpp
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 < 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 `prev->suffix().length()` is not 0 the operator sets `*this` to a suffix iterator that points to the range `[prev->suffix().first, prev->suffix().second)`. Otherwise, sets `*this` to an end-of-sequence iterator.
6
7
Returns: `*this`
```

```cpp
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.

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.

The following productions within the ECMAscript grammar are modified as follows:
The following new productions are then added:

```
ClassAtomExClass ::
    [: ClassName :]
ClassAtomCollatingElement ::
    [. ClassName .]
ClassAtomEquivalence ::
    [= ClassName =]
```

The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in regular expressions in POSIX.

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 124.

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 `ClassName`s 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 "uxxxx" that do not fit in a character are invalid. — end note]

Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling traits_inst.value.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags (30.5.2) 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())
&& 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 below.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.3 Type aliases</td>
<td></td>
</tr>
<tr>
<td>31.4 Order and consistency</td>
<td></td>
</tr>
<tr>
<td>31.5 Lock-free property</td>
<td></td>
</tr>
<tr>
<td>31.6 Class template atomic_ref &lt;atomic&gt;</td>
<td></td>
</tr>
<tr>
<td>31.7 Class template atomic &lt;atomic&gt;</td>
<td></td>
</tr>
<tr>
<td>31.8 Non-member functions &lt;atomic&gt;</td>
<td></td>
</tr>
<tr>
<td>31.9 Flag type and operations &lt;atomic&gt;</td>
<td></td>
</tr>
<tr>
<td>31.10 Fences &lt;atomic&gt;</td>
<td></td>
</tr>
</tbody>
</table>

31.2 Header <atomic> synopsis

```cpp
namespace std {
    enum class memory_order : unspecified;
    template<class T>
        T kill_dependency(T y) noexcept;

    #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

    template<class T> struct atomic_ref;
    template<class T> struct atomic_ref<T*>;

    template<class T> struct atomic;
    template<class T> struct atomic<T*>;

    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;
```

§ 31.2
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;

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>
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>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_add(atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_add_explicit(volatile atomic<T>*, typename atomic<T>::difference_type,
memory_order) noexcept;
template<class T>
T atomic_fetch_add_explicit(atomic<T>*, typename atomic<T>::difference_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_sub(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_sub(atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_sub_explicit(volatile atomic<T>*, typename atomic<T>::difference_type,
memory_order) noexcept;
template<class T>
T atomic_fetch_sub_explicit(atomic<T>*, typename atomic<T>::difference_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_and(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_and(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_and_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;
template<class T>
T atomic_fetch_and_explicit(atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_or(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_or(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_or_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;
template<class T>
T atomic_fetch_or_explicit(atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

template<class T>
T atomic_fetch_xor(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_xor(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_xor_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;
template<class T>
T atomic_fetch_xor_explicit(atomic<T>*, typename atomic<T>::value_type,
memory_order) noexcept;

// 31.7.1, initialization
#define ATOMIC_VAR_INIT(value)
see below

// 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_ulonglong = 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_int_fast64_t = atomic<int_fast64_t>;
using atomic_uint_fast64_t = atomic<uint_fast64_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>;

// 31.9, flag type and operations
struct atomic_flag;
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;
define ATOMIC_FLAG_INIT see below

// 31.10, 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_int N_t`, `atomic_uint N_t`, `atomic_intptr_t`, and `atomic_uintptr_t` are defined if and only if `int N_t`, `uint N_t`, `intptr_t`, and `uintptr_t` are defined, respectively.

31.4 Order and consistency

```cpp
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 6.8.2 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

1. `memory_order::relaxed`: no operation orders memory.
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.
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]
4. `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]

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`.

An atomic operation `A` on some atomic object `M` is coherence-ordered before another atomic operation `B` on `M` if

1. `A` is a modification, and `B` reads the value stored by `A`, or
2. `A` precedes `B` in the modification order of `M`, or
3. `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
4. there exists an atomic modification `X` of `M` such that `A` is coherence-ordered before `X` and `X` is coherence-ordered before `B`.

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`:

1. if `A` and `B` are both `memory_order::seq_cst` operations, then `A` precedes `B` in `S`; and
2. 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
3. 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
4. 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`. 

§ 31.4 1481
Note: This definition ensures that \( S \) is consistent with the modification order of any atomic object \( M \). It also ensures that a \texttt{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-\texttt{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” (6.8.2.1). This allows more efficient implementation of \texttt{memory\_order::acquire} and \texttt{memory\_order::release} on some machine architectures. It can produce surprising results when these are mixed with \texttt{memory\_order::seq\_cst} accesses. —end note

Note: \texttt{memory\_order::seq\_cst} ensures sequential consistency only for a program that is free of data races and uses exclusively \texttt{memory\_order::seq\_cst} atomic operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In many cases, \texttt{memory\_order::seq\_cst} atomic operations are reorderable with respect to other atomic operations performed by the same thread. —end note

Implementations should ensure that no “out-of-thin-air” values are computed that circularly depend on their own computation.

Note: For example, with \( x \) and \( y \) initially zero,

\begin{verbatim}
// 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);
\end{verbatim}

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

Note: The recommendation similarly disallows \( r1 == r2 == 42 \) in the following example, with \( x \) and \( y \) again initially zero:

\begin{verbatim}
// 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
\end{verbatim}

Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

\begin{verbatim}
template<class T>
T kill_dependency(T y) noexcept;
\end{verbatim}

Effects: The argument does not carry a dependency to the return value (6.8.2).

Returns: \( y \).

31.5 Lock-free property

The \texttt{ATOMIC\_\ldots\_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 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.

2 The function atomic_is_lock_free (31.7.1) 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 (6.8.2.2).

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 Class template atomic_ref

```cpp
namespace std {
    template<class T> struct atomic_ref {
        private:
            T* ptr; // exposition only
        public:
            using value_type = T;
            static constexpr bool is_always_lock_free = implementation-defined;
            static constexpr size_t required_alignment = implementation-defined;

            atomic_ref& operator=(const atomic_ref&) = delete;
        explicit atomic_ref(T&); // exposition only
        atomic_ref(const atomic_ref&) noexcept;

        T operator=(T) const noexcept;
        operator T() const noexcept;
        bool is_lock_free() const noexcept;
        void store(T, memory_order = memory_order_seq_cst) const noexcept;
        T load(memory_order = memory_order_seq_cst) 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;
    };
}
```

1 An atomic_ref object applies atomic operations (31.1) to the object referenced by *ptr such that, for the lifetime (6.6.3) of the atomic_ref object, the object referenced by *ptr is an atomic object (6.8.2.1).

2 The template argument for T shall be trivially copyable (6.7).

3 The lifetime (6.6.3) 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.

4 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]

31.6.1 Operations

```cpp
static constexpr bool is_always_lock_free;

1 The static data member is_always_lock_free is true if the atomic_ref type’s operations are always lock-free, and false otherwise.
```
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 (6.6.6)
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]

atomic_ref(T& obj);

Requires: The referenced object shall be aligned to required_alignment.
Effects: Constructs an atomic reference that references the object.
Throws: Nothing.

atomic_ref(const atomic_ref& ref) noexcept;

Effects: Constructs an atomic reference that references the object referenced by ref.

T operator=(T desired) const noexcept;

Effects: Equivalent to:
store(desired);
return desired;

operator T() const noexcept;

Effects: Equivalent to: return load();

bool is_lock_free() const noexcept;

Returns: true if the object's operations are lock-free, false otherwise.

void store(T desired, memory_order order = memory_order_seq_cst) const noexcept;

Requires: The order argument shall not be 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.

T load(memory_order order = memory_order_seq_cst) const noexcept;

Requires: The order argument shall not be 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.

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 (6.8.2).

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;

Requires: The failure argument shall not be 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 (6.8.2.1) 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]

31.6.2 Specializations for integral types [atomics.ref.int]

There are specializations of the `atomic_ref` class template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, and any other types needed by the typedefs in the header `<cstdint>`. For each such type `integral`, the specialization `atomic_ref<integral>` provides additional atomic operations appropriate to integral types. [Note: For the specialization `atomic_ref<bool>`, see 31.6. —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 bool is_always_lock_free = implementation-defined;
            static constexpr size_t required_alignment = implementation-defined;

            atomic_ref& operator=(const atomic_ref&) = delete;
            explicit atomic_ref(integral&);
            atomic_ref(const atomic_ref&) noexcept;
            integral operator=(integral) const noexcept;
            operator integral() const noexcept;

            bool is_lock_free() const noexcept;
            void store(integral, memory_order = memory_order_seq_cst) const noexcept;
            integral load(memory_order = memory_order_seq_cst) 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;

integral operator op=(integral operand) const noexcept;

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 (6.8.2.1).

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]

integral operator op=(integral operand) const noexcept;

Effects: Equivalent to: return fetch_key(operand) op operand;

31.6.3 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.

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 bool is_always_lock_free = implementation-defined;
            static constexpr size_t required_alignment = implementation-defined;

            atomic_ref& operator=(const atomic_ref&) = delete;

            explicit atomic_ref(floating-point);
            atomic_ref(const atomic_ref&) noexcept;

            floating-point operator=(floating-point) noexcept;
            operator floating-point() const noexcept;
    };
}
bool is_lock_free() const noexcept;
void store(floating-point, memory_order = memory_order_seq_cst) const noexcept;
floating-point load(memory_order = memory_order_seq_cst) const noexcept;
floating-point exchange(floating-point,
memory_order = memory_order_seq_cst) const noexcept;
bool compare_exchange_weak(floating-pointk, floating-point,
memory_order, memory_order) const noexcept;
bool compare_exchange_strong(floating-pointk, floating-point,
memory_order, memory_order) const noexcept;
bool compare_exchange_weak(floating-pointk, floating-point,
memory_order, memory_order) const noexcept;
bool compare_exchange_strong(floating-pointk, floating-point,
memory_order = memory_order_seq_cst) const noexcept;

floating-point fetch_add(floating-point,
memory_order = memory_order_seq_cst) const noexcept;
floating-point fetch_sub(floating-point,
memory_order = memory_order_seq_cst) const noexcept;

floating-point operator+=(floating-point) const noexcept;
floating-point operator-=(floating-point) const noexcept;
};

2 Descriptions are provided below only for members that differ from the primary template.

3 The following operations perform arithmetic computations. The key, operator, and computation correspondence are identified in Table 132.

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 (6.8.2.1).

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 (17.3.2). 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 op=(floating-point operand) const noexcept;

Effects: Equivalent to: return fetch_key(operand) op operand;

31.6.4 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 bool is_always_lock_free = implementation-defined;
            static constexpr size_t required_alignment = implementation-defined;

            atomic_ref& operator=(const atomic_ref&) = delete;
            explicit atomic_ref(T*&) = default;
            atomic_ref(const atomic_ref&) noexcept;

            T* operator=(T*) const noexcept;
            operator T*() const noexcept;
        };
    }
}
true is_lock_free() const noexcept;
void store(T*, memory_order = memory_order_seq_cst) const noexcept;
T* load(memory_order = memory_order_seq_cst) 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;

1 Descriptions are provided below only for members that differ from the primary template.
2 The following operations perform arithmetic computations. The key, operator, and computation correspondence is identified in Table 133.

T* fetch_key(difference_type operand, memory_order = memory_order_seq_cst) const noexcept;

3 Requires: T shall be an object type, otherwise the program is ill-formed.
4 Effects: Atomically replaces the value referenced by *ptr with the result of the computation applied to the value referenced by *ptr and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (6.8.2.1).
5 Returns: Atomically, the value referenced by *ptr immediately before the effects.
6 Remarks: The result may be an undefined address, but the operations otherwise have no undefined behavior.

T* operator op=(difference_type operand) const noexcept;

7 Effects: Equivalent to: return fetch_key(operand) op operand;

31.6.5 Member operators common to integers and pointers to objects
[atomics.ref.memop]

T* operator++(int) const noexcept;

1 Effects: Equivalent to: return fetch_add(1);

T* operator--(int) const noexcept;

2 Effects: Equivalent to: return fetch_sub(1);

T* operator++() const noexcept;

3 Effects: Equivalent to: return fetch_add(1) + 1;

T* operator--(int) const noexcept;

4 Effects: Equivalent to: return fetch_sub(1) - 1;

31.7 Class template atomic

[atomics.types.generic]

namespace std {
    template<class T> struct atomic {
        using value_type = T;

    };

§ 31.7 1488
static constexpr bool is_always_lock_free = implementation-defined;
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
void store(T, memory_order = memory_order::seq_cst) volatile noexcept;
void store(T, memory_order = memory_order::seq_cst) 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;
T load(memory_order = memory_order::seq_cst) volatile noexcept;
T load(memory_order = memory_order::seq_cst) noexcept;
operator T() volatile noexcept;
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;
atomic() noexcept = default;
constexpr atomic(T) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;
T operator=(T) volatile noexcept;
T operator=(T) 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]

31.7.1 Operations on atomic types [atomics.types.operations]

1 [Note: Many operations are volatile-qualified. The “volatile as device register” semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. — end note]

atomic() noexcept = default;

2 Effects: Leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatibility with C. — end note]

constexpr atomic(T desired) noexcept;

3 Effects: Initializes the object with the value desired. Initialization is not an atomic operation (6.8.2). [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]
#define ATOMIC_VAR_INIT(value) see below

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:

atomic<int> v = ATOMIC_VAR_INIT(5);
—end example]

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;

Requires: The order argument shall not be memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.

Effects: Atomically replaces the value pointed to by this with the value of desired. Memory is affected according to the value of order.

T operator=(T desired) volatile noexcept;
T operator=(T desired) noexcept;

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;

Requires: The order argument shall not be memory_order::release nor memory_order::acq_rel.

Effects: Memory is affected according to the value of order.

Returns: Atomically returns the value pointed to by this.

operator T() const volatile noexcept;
operator T() const noexcept;

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;

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 (6.8.2).

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 noexcep;
bool compare_exchange_strong(T& expected, T desired,
memory_order success, memory_order failure) noexcep;

bool compare_exchange_weak(T& expected, T desired,
memory_order order = memory_order::seq_cst) volatile noexcep;
bool compare_exchange_weak(T& expected, T desired,
    memory_order order = memory_order::seq_cst) noexcept;
bool compare_exchange_strong(T& expected, T desired,
    memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
    memory_order order = memory_order::seq_cst) noexcept;

Requires: The failure argument shall not be memory_order::release nor memory_order::acq_rel.

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 (6.8.2) 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 (6.7) is
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.
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:
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:

```cpp
struct padded {
    char clank = 0x42;  // Padding here.
    unsigned biff = 0xC0DEFEFE;
};
atomic<padded> pad = ATOMIC_VAR_INIT({});

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:

```cpp
union pony {
    double celestia = 0.;
    short luna;  // padded
};
atomic<pony> princesses = ATOMIC_VAR_INIT({});

bool party(pony desired) {
    pony expected;
    return princesses.compare_exchange_strong(expected, desired);
}
```

— end note]

### 31.7.2 Specializations for integers

There are specializations of the `atomic` class template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `char8_t`, `char16_t`, `char32_t`, and any other types needed by the typedefs in the header `<cstdint>`. For each such type `integral`, the specialization `atomic<integral>` provides additional atomic
operations appropriate to integral types. [Note: For the specialization `atomic<bool>`, see 31.7. — end note]

```cpp
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;
        void store(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        void store(integral, memory_order = memory_order::seq_cst) noexcept;
        integral load(memory_order = memory_order::seq_cst) const volatile noexcept;
        integral load(memory_order = memory_order::seq_cst) const noexcept;
        operator integral() const volatile noexcept;
        operator integral() const noexcept;
        integral exchange(integral, memory_order = memory_order::seq_cst) volatile noexcept;
        integral exchange(integral, memory_order = memory_order::seq_cst) noexcept;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order, memory_order) volatile noexcept;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order, memory_order) noexcept;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order, memory_order) volatile noexcept;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order, memory_order) noexcept;
    }
}
```
bool compare_exchange_weak(integral&, integral,
    memory_order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_weak(integral&, integral,
    memory_order = memory_order::seq_cst) noexcept;
bool compare_exchange_strong(integral&, integral,
    memory_order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(integral&, integral,
    memory_order = memory_order::seq_cst) noexcept;

integral fetch_add(integral, memory_order = memory_order::seq_cst) volatile noexcept;
integral fetch_add(integral, memory_order = memory_order::seq_cst) noexcept;
integral fetch_sub(integral, memory_order = memory_order::seq_cst) volatile noexcept;
integral fetch_sub(integral, memory_order = memory_order::seq_cst) noexcept;
integral fetch_and(integral, memory_order = memory_order::seq_cst) volatile noexcept;
integral fetch_and(integral, memory_order = memory_order::seq_cst) noexcept;
integral fetch_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;

atomic() noexcept = default;
constexpr atomic(integral) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;

integral operator=(integral) volatile noexcept;
integral operator=(integral) noexcept;
integral operator++(int) volatile noexcept;
integral operator++(int) noexcept;
integral operator--(int) volatile noexcept;
integral operator--(int) noexcept;
integral operator++() volatile noexcept;
integral operator++() noexcept;
integral operator--() volatile noexcept;
integral operator--() noexcept;
integral operator+=(integral) volatile noexcept;
integral operator+=(integral) 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;

T fetch_key(T operand, memory_order order = memory_order::seq_cst) volatile noexcept;

---

2 The atomic integral specializations are standard-layout structs. They each have a trivial default constructor and 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>
<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>
T fetch_key(T operand, memory_order order = memory_order::seq_cst) noexcept;

5 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 (6.8.2).

6 Returns: Atomically, the value pointed to by this immediately before the effects.

7 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]

T operator op=(T operand) volatile noexcept;
T operator op=(T operand) noexcept;

8 Effects: Equivalent to: return fetch_key(operand) op operand;

31.7.3 Specializations for floating-point types [atomics.types.float]

There are specializations of the atomic class template for the floating-point types float, double, and long double. For each such type floating-point, the specialization atomic-floating-point provides additional atomic operations appropriate to floating-point types.

namespace std {
    template<> struct atomic<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;
        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 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;
        bool compare_exchange_weak(floating-point&, floating-point,
            memory_order = memory_order_seq_cst) volatile noexcept;
        bool compare_exchange_weak(floating-point&, floating-point,
            memory_order = memory_order_seq_cst) noexcept;
        bool compare_exchange_strong(floating-point&, floating-point,
            memory_order = memory_order_seq_cst) volatile noexcept;
        bool compare_exchange_strong(floating-point&, floating-point,
            memory_order = memory_order_seq_cst) 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;
    };

§ 31.7.3
The atomic floating-point specializations are standard-layout structs. They each have a trivial default constructor and 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 132.

\[
\text{T A::fetch_key(T operand, memory\_order order = memory\_order\_seq\_cst) volatile noexcept;}
\]

Effects: Atomically replaces the value pointed to by \textit{this} with the result of the computation applied to the value pointed to by \textit{this} and the given \textit{operand}. Memory is affected according to the value of \textit{order}. These operations are atomic read-modify-write operations (6.8.2).

Returns: Atomically, the value pointed to by \textit{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 \textit{floating-point} should conform to the \textit{std::numeric\_limits<\textit{floating-point}>} traits associated with the floating-point type (17.3.2). The floating-point environment (26.3) for atomic arithmetic operations on \textit{floating-point} may be different than the calling thread’s floating-point environment.

\[
\text{T operator op=(T operand) volatile noexcept;}
\]

Effects: Equivalent to: \text{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 \textit{floating-point} should conform to the \textit{std::numeric\_limits<\textit{floating-point}>} traits associated with the floating-point type (17.3.2). The floating-point environment (26.3) for atomic arithmetic operations on \textit{floating-point} may be different than the calling thread’s floating-point environment.

31.7.4 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;
        void store(T*, memory_order = memory_order::seq_cst) volatile noexcept;
        void store(T*, memory_order = memory_order::seq_cst) 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*, 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;
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;
atomic() noexcept = default;
constexpr atomic(T*) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;
T* operator=(T*) volatile noexcept;
T* operator=(T*) 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;

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 default constructor and 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 Requires: T shall be an object type, otherwise the program is ill-formed. [Note: Pointer arithmetic on void* or function pointers is ill-formed. —end note]

5 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 (6.8.2).

6 Returns: Atomically, the value pointed to by this immediately before the effects.

7 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;

Effects: Equivalent to: return fetch_key(operand) op operand;

31.7.5 Member operators common to integers and pointers to objects
[atomics.types.memop]

T operator++(int) volatile noexcept;
T operator++(int) noexcept;

Effects: Equivalent to: return fetch_add(1);

T operator--(int) volatile noexcept;
T operator--(int) noexcept;

Effects: Equivalent to: return fetch_sub(1);

T operator++() volatile noexcept;
T operator++() noexcept;

Effects: Equivalent to: return fetch_add(1) + 1;

T operator--() volatile noexcept;
T operator--() noexcept;

Effects: Equivalent to: return fetch_sub(1) - 1;

31.8 Non-member functions
[atomics.nonmembers]

A non-member function template whose name matches the pattern atomic_f or the pattern atomic_f_-explicit invokes the member function 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 atomic<T>::value_type* is dereferenced when passed to the member function call. If no such member function exists, the program is ill-formed.

template<class T>
void atomic_init(volatile atomic<T>* object, typename atomic<T>::value_type desired) noexcept;

Effects: Non-atomically initializes *object with value desired. This function shall only be applied to objects that have been default constructed, and then only once. [Note: These semantics ensure compatibility with C. —end note] [Note: Concurrent access from another thread, even via an atomic operation, constitutes a data race. —end note]

[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. —end note]

31.9 Flag type and operations
[atomics.flag]

namespace std {

struct atomic_flag {
    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;

    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;
};

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;

§ 31.9
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;

#define ATOMIC_FLAG_INIT see below
}

1. The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear.

2. Operations on an object of type `atomic_flag` shall be lock-free. [Note: Hence the operations should also be address-free. — end note]

3. The `atomic_flag` type is a standard-layout struct. It has a trivial default constructor and a trivial destructor.

4. The macro `ATOMIC_FLAG_INIT` shall be 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. Unless initialized with `ATOMIC_FLAG_INIT`, it is unspecified whether an `atomic_flag` object has an initial state of set or clear.

5. `bool atomic_flag_test_and_set(volatile atomic_flag* object) noexcept;
bool atomic_flag_test_and_set(atomic_flag* object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag* object, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag* object, memory_order order) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) noexcept;`

   **Effects:** Atomically sets the value pointed to by `object` or by `this` to `true`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (6.8.2).

   **Returns:** Atomically, the value of the object immediately before the effects.

7. `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;`

   **Requires:** The `order` argument shall not be `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

   **Effects:** Atomically sets the value pointed to by `object` or by `this` to `false`. Memory is affected according to the value of `order`.

31.10 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*.

2. 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.

3. 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.

4. 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`.

5. `extern "C" void atomic_thread_fence(memory_order order) noexcept;`

   **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 (6.8.2), perform mutual exclusion, and communicate conditions and values between threads, as summarized in Table 134.

Table 134 — Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.2</td>
<td></td>
</tr>
<tr>
<td>32.3</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>32.4</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>32.5</td>
<td>&lt;condition_variable&gt;</td>
</tr>
<tr>
<td>32.6</td>
<td>&lt;future&gt;</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 shall be 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 shall be reported as described in 16.5.5.12.

[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_code` 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 shall compare 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
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.\(^{330}\) Given a duration argument \(D_t\), the real-time duration of the timeout is \(D_t + D_i + D_m\).

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. \([\text{Note:} \text{This specification may result in the total duration of the wait decreasing when measured against a steady clock.}]\)

\(^{(4.2)}\) — if \(C_a \leq C_t\), the waiting function should not time out until \(\text{Clock::now()}\) returns a time \(C_n \geq C_t\), i.e., waking at \(C_t + D_i + D_m\). \([\text{Note:} \text{When the clock is adjusted backwards, this specification may result in the total duration of the wait increasing when measured against a steady clock. When the clock is adjusted forwards, this specification may result in the total duration of the wait decreasing when measured against a steady clock.}]\)

An implementation shall return 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. \([\text{Note:} \text{Implementations should decrease the duration of the wait when the clock is adjusted forwards.}]\)

\(^{5}\) [\text{Note:} \text{If the clock is not synchronized with a steady clock, e.g., a CPU time clock, these timeouts might not provide useful functionality.}]\)

\(^{6}\) 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 shall satisfy 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. \([\text{Note:} \text{Instantiations of clock, time point and duration types supplied by the implementation as specified in 27.7 do not throw exceptions.}]\)

### 32.2.5 Requirements for Cpp17Lockable types

#### 32.2.5.1 In general

An execution agent is an entity such as a thread that may perform work in parallel with other execution agents. \([\text{Note:} \text{Implementations or users may introduce other kinds of agents such as processes or thread-pool tasks.}]\) The calling agent is determined by context, e.g., the calling thread that contains the call, and so on.

\(^{1}\) [\text{Note:} \text{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).}]\)

\(^{3}\) The standard library templates unique_lock (32.4.4.3), shared_lock (32.4.4.4), scoped_lock (32.4.4.2), lock_guard (32.4.4.1), lock, try_lock (32.4.5), and condition_variable_any (32.5.4) 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. \([\text{Note:} \text{The nature of any lock ownership and any synchronization it may entail are not part of these requirements.}]\)

#### 32.2.5.2 Cpp17BasicLockable requirements

A type \(L\) meets the Cpp17BasicLockable requirements if the following expressions are well-formed and have the specified semantics (\(m\) denotes a value of type \(L\)).

\(^{330}\) All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.
m.lock()

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()

Requires: The current execution agent shall hold a lock on m.

Effects: Releases a lock on m held by the current execution agent.

Throws: Nothing.

32.2.5.3 Cpp17Lockable requirements

A type L meets the Cpp17Lockable requirements if it meets the Cpp17BasicLockable requirements and the following expressions are well-formed and have the specified semantics (m denotes a value of type L).

m.try_lock()

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.

Return type: bool.

Returns: true if the lock was acquired, false otherwise.

32.2.5.4 Cpp17TimedLockable requirements

A type L meets the Cpp17TimedLockable requirements if it meets the Cpp17Lockable requirements and the following expressions are well-formed and have the specified semantics (m denotes a value of type L, 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)

Effects: Attempts to acquire a lock for the current execution agent within the relative timeout (32.2.4) specified by rel_time. The function shall 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 shall not have been acquired for the current execution agent.

Return type: bool.

Returns: true if the lock was acquired, false otherwise.

m.try_lock_until(abs_time)

Effects: Attempts to acquire a lock for the current execution agent before the absolute timeout (32.2.4) specified by abs_time. The function shall 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 shall not have been acquired for the current execution agent.

Return type: bool.

Returns: true if the lock was acquired, false otherwise.

32.3 Threads

32.3 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]

32.3.1 Header <thread> synopsis

namespace std {
    class thread {
        void swap(thread& x, thread& y) noexcept;

        namespace this_thread {
            thread::id get_id() noexcept;
        }
    }
}

§ 32.3.1
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);
}
}

32.3.2 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;
  };
}

32.3.2.1 Class thread::id

namespace std {
  class thread::id {
    public:
    id() noexcept;
  };

bool operator==(thread::id x, thread::id y) noexcept;
bool operator!=(thread::id x, thread::id y) noexcept;
bool operator<(thread::id x, thread::id y) noexcept;
bool operator>(thread::id x, thread::id y) noexcept;
bool operator<=(thread::id x, thread::id y) noexcept;
bool operator>=(thread::id x, thread::id y) noexcept;

§ 32.3.2.1

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 (32.3.2). 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.

`thread::id` is a trivially copyable class (11.1). The library may reuse the value of a `thread::id` of a terminated thread that can no longer be joined.

[Note: Relational operators allow `thread::id` objects to be used as keys in associative containers. — end note]

Effects: Constructs an object of type `id`.

Ensures: The constructed object does not represent a thread of execution.

Returns: `true` only if `x` and `y` represent the same thread of execution or neither `x` nor `y` represents a thread of execution.

Returns: A value such that `operator<` is a total ordering as described in 25.7.

Returns: `y < x`.

Returns: `!(y < x)`.

Returns: `!(x < y)`.

Effects: Inserts an unspecified text representation of `id` into `out`. For two objects of type `thread::id` `x` and `y`, if `x == y` the `thread::id` objects have the same text representation and if `x != y` the `thread::id` objects have distinct text representations.

Returns: `out`.

The specialization is enabled (20.14.18).

### 32.3.2.2 Constructors

**Effects:** Constructs a `thread` object that does not represent a thread of execution.

**Ensures:** `get_id()` == `id()`.
template<class F, class... Args> explicit thread(F&& f, Args&&... args);

Requires: F and each Ti in Args shall satisfy the Cpp17MoveConstructible requirements. INVOKE(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...) (20.14.3) shall be a valid expression.

Remarks: This constructor shall not participate in overload resolution if remove_cvref_t<F> is the same type as std::thread.

Effects: Constructs an object of type thread. 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(decay-copy(std::forward<F>(f)), decay-copy(std::forward<Args>(args))...) terminates with an uncaught exception, terminate shall be called.

Synchronization: The completion of the invocation of the constructor synchronizes with the beginning of the invocation of the copy of f.

Ensures: get_id() != id(). *this represents the newly started thread.

Throws: system_error if unable to start the new thread.

Error conditions:

resource_unavailable_try_again — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

thread(thread&& x) noexcept;

Effects: Constructs an object of type thread from x, and sets x to a default constructed state.

Ensures: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the start of construction.

32.3.2.3 Destructor

~thread();

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]

32.3.2.4 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.

Ensures: x.get_id() == id() and get_id() returns the value of x.get_id() prior to the assignment.

Returns: *this.

32.3.2.5 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 (6.8.2) the corresponding successful join() return. [Note: Operations on *this are not synchronized. —end note]
Ensures: 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 shall release any owned resources.

Ensures: 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.

32.3.2.6 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.

32.3.2.7 Specialized algorithms

void swap(thread& x, thread& y) noexcept;

Effects: As if by x.swap(y).

32.3.3 Namespace this_thread

namespace std::this_thread {

thread::id get_id() noexcept;

void yield() noexcept;

template<class Clock, class Duration>
void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);

template<class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& rel_time);

}

thread::id this_thread::get_id() noexcept;

Returns: An object of type thread::id that uniquely identifies the current thread of execution. No other thread of execution shall have this id and this thread of execution shall always have this id. The object returned shall 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).

32.4 Mutual exclusion

This subclause provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (6.8.2).

32.4.1 Header <mutex> synopsis

namespace std {

class mutex;
class recursive_mutex;
class timed_mutex;
class recursive_timed_mutex;

struct defer_lock_t { explicit defer_lock_t() = default; private; }
struct try_to_lock_t { explicit try_to_lock_t() = default; private; }
struct adopt_lock_t { explicit adopt_lock_t() = default; private; }

inline constexpr defer_lock_t defer_lock { }; // default
inline constexpr try_to_lock_t try_to_lock { }; // default
inline constexpr adopt_lock_t adopt_lock { }; // default

template<class Mutex> class lock_guard;
template<class... MutexTypes> class scoped_lock;
template<class Mutex> class unique_lock;

template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

struct once_flag;

template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);
}

32.4.2 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;
}

§ 32.4.2
32.4.3 Mutex requirements

32.4.3.1 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.

32.4.3.2 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 shall satisfy the requirements set out in this subclause. In this description, m denotes an object of a mutex type.

The mutex types shall satisfy the Cpp17Lockable requirements (32.2.5.3).

The mutex types shall be Cpp17DefaultConstructible and Cpp17Destructible. If initialization of an object of a mutex type fails, an exception of type system_error shall be thrown. The mutex types shall not be copyable or movable.

The error conditions for error codes, if any, reported by member functions of the mutex types shall be:

- resource_unavailable_try_again — if any native handle type manipulated is not available.
- operation_not_permitted — if the thread does not have the privilege to perform the operation.
- invalid_argument — if any native handle type manipulated as part of mutex construction is incorrect.

The implementation shall provide lock and unlock operations, as described below. For purposes of determining the existence of a data race, these behave as atomic operations (6.8.2). The lock and unlock operations on a single mutex shall appear to occur in a single total order. [Note: This can be viewed as the modification order (6.8.2) 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]

The expression m.lock() shall be well-formed and have the following semantics:

**Requires:** 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.

**Ensures:** The calling thread owns the mutex.

**Return type:** void.

**Synchronization:** Prior unlock() operations on the same object shall synchronize with (6.8.2) this operation.

**Throws:** system_error when an exception is required (32.2.2).

**Error conditions:**

- operation_not_permitted — if the thread does not have the privilege to perform the operation.
- resource_deadlock_would_occur — if the implementation detects that a deadlock would occur.

The expression m.try_lock() shall be well-formed and have the following semantics:

**Requires:** If m is of type mutex, timed_mutex, shared_mutex, or shared_timed_mutex, the calling thread does not own the mutex.

**Effects:** Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and try_lock() immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [Note: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (Clause 31). —end note] An implementation should ensure that try_lock() does not consistently return false in the absence of contending mutex acquisitions.

**Return type:** bool.

**Returns:** true if ownership of the mutex was obtained for the calling thread, otherwise false.

**Synchronization:** If try_lock() returns true, prior unlock() operations on the same object synchronize with (6.8.2) this operation. [Note: Since lock() does not synchronize with a failed subsequent try_lock()...
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]

20 Throws: Nothing.

21 The expression \( m.\text{unlock()} \) shall be well-formed and have the following semantics:

22 Requires: The calling thread shall own the mutex.

23 Effects: Releases the calling thread’s ownership of the mutex.

24 Return type: void.

25 Synchronization: This operation synchronizes with (6.8.2) subsequent lock operations that obtain ownership on the same object.

26 Throws: Nothing.

32.4.3.2.1 Class mutex

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
    };
}

1 The class mutex provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for \( \text{try\_lock()} \)) or block (for \( \text{lock()} \)) until the owning thread has released ownership with a call to \( \text{unlock()} \).

2 [Note: After a thread \( A \) has called \( \text{unlock()} \), releasing a mutex, it is possible for another thread \( B \) to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread \( A \) appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread \( A \) doesn’t access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. —end note]

3 The class mutex shall satisfy all of the mutex requirements (32.4.3). It shall be a standard-layout class (11.1).

4 [Note: A program may deadlock if the thread that owns a mutex object calls \( \text{lock()} \) on that object. If the implementation can detect the deadlock, a resource\_deadlock\_would\_occur error condition may 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.

32.4.3.2.2 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();

§ 32.4.3.2.2 1509
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.

The class `recursive_mutex` shall satisfy all of the mutex requirements (32.4.3). It shall be a standard-layout class (11.1).

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()` shall fail, and additional calls to `lock()` shall 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.

The behavior of a program is undefined if:

1. It destroys a `recursive_mutex` object owned by any thread or
2. A thread terminates while owning a `recursive_mutex` object.

### 32.4.3.3 Timed mutex types

The timed mutex types are the standard library types `timed_mutex`, `recursive_timed_mutex`, and `shared_timed_mutex`. They shall meet the requirements set out below. In this description, `m` denotes an object of a mutex type, `rel_time` denotes an object of an instantiation of `duration` (27.5), and `abs_time` denotes an object of an instantiation of `time_point` (27.6).

The timed mutex types shall satisfy the `Cpp17TimedLockable` requirements (32.2.5.4).

The expression `m.try_lock_for(rel_time)` shall be well-formed and have the following semantics:

**Requires:** 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 shall return 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 `try_lock_for()` returns `true`, prior `unlock()` operations on the same object synchronize with (6.8.2) this operation.

**Throws:** Timeout-related exceptions (32.2.4).

The expression `m.try_lock_until(abs_time)` shall be well-formed and have the following semantics:

**Requires:** If `m` is of type `timed_mutex` or `shared_timed_mutex`, the calling thread does not own the mutex.

**Effects:** The function attempts to obtain ownership of the mutex. If `abs_time` has already passed, the function attempts to obtain ownership without blocking (as if by calling `try_lock()`). The function shall return before the absolute timeout (32.2.4) specified by `abs_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 `try_lock_until()` returns `true`, prior `unlock()` operations on the same object synchronize with (6.8.2) this operation.
§ 32.4.3.3.1 Class `timed_mutex`

```
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
    }
}
```

1 The class `timed_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a `timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()` or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has released ownership with a call to `unlock()` or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

2 The class `timed_mutex` shall satisfy all of the timed mutex requirements (32.4.3.3). It shall be a standard-layout class (11.1).

3 The behavior of a program is undefined if:
   
   (3.1) it destroys a `timed_mutex` object owned by any thread,
   
   (3.2) a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object, or
   
   (3.3) a thread terminates while owning a `timed_mutex` object.

§ 32.4.3.3.2 Class `recursive_timed_mutex`

```
namespace std {
    class recursive_timed_mutex {
        public:
            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
    }
}
```
The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has completely released ownership or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

The class `recursive_timed_mutex` shall satisfy all of the timed mutex requirements (32.4.3.3). It shall be a standard-layout class (11.1).

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()` shall fail, and additional calls to `lock()` shall throw an exception of type `system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()`, `try_lock()`, `try_lock_for()`, and `try_lock_until()`. Only when all levels of ownership have been released may ownership of the object be acquired by another thread.

The behavior of a program is undefined if:

1. it destroys a `recursive_timed_mutex` object owned by any thread, or
2. a thread terminates while owning a `recursive_timed_mutex` object.

### 32.4.3.4 Shared mutex types

The standard library types `shared_mutex` and `shared_timed_mutex` are shared mutex types. Shared mutex types shall satisfy the requirements of mutex types (32.4.3.2), and additionally shall satisfy the requirements set out below. In this description, \( m \) denotes an object of a shared mutex type.

In addition to the exclusive lock ownership mode specified in 32.4.3.2, 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 shall hold 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 shall be at least 10000. If more than the maximum number of execution agents attempt to obtain a shared lock, the excess execution agents shall block until the number of shared locks are reduced below the maximum amount by other execution agents releasing their shared lock.

The expression \( m.lock_shared() \) shall be well-formed and have the following semantics:

1. **Requires**: The calling thread has no ownership of the mutex.
2. **Effects**: Blocks the calling thread until shared ownership of the mutex can be obtained for the calling thread. If an exception is thrown then a shared lock shall not have been acquired for the current thread.
3. **Ensures**: The calling thread has a shared lock on the mutex.
4. **Return type**: `void`.
5. **Synchronization**: Prior `unlock()` operations on the same object shall synchronize with (6.8.2) this operation.
6. **Throws**: `system_error` when an exception is required (32.2.2).

The expression \( m.unlock_shared() \) shall be well-formed and have the following semantics:

1. **Requires**: The calling thread shall hold a shared lock on the mutex.
2. **Effects**: Releases a shared lock on the mutex held by the calling thread.
3. **Return type**: `void`.
4. **Synchronization**: This operation synchronizes with (6.8.2) subsequent `lock()` operations that obtain ownership on the same object.
5. **Throws**: Nothing.

---

© ISO/IEC N4810

§ 32.4.3.4 1512
The expression `m.try_lock_shared()` shall be well-formed and have the following semantics:

- **Requires**: 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 (6.8.2) this operation.
- **Throws**: Nothing.

### 32.4.3.4.1 Class `shared_mutex` [thread.sharedmutex.class]

```cpp
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
    };
}
```

1. The class `shared_mutex` provides a non-recursive mutex with shared ownership semantics.
2. The class `shared_mutex` shall satisfy all of the shared mutex requirements (32.4.3.4). It shall be a standard-layout class (11.1).
3. The behavior of a program is undefined if:
   1. it destroys a `shared_mutex` object owned by any thread,
   2. a thread attempts to recursively gain any ownership of a `shared_mutex`, or
   3. a thread terminates while possessing any ownership of a `shared_mutex`.
4. `shared_mutex` may be a synonym for `shared_timed_mutex`.

### 32.4.3.5 Shared timed mutex types [thread.sharedtimedmutex.requirements]

1. The standard library type `shared_timed_mutex` is a `shared timed mutex type`. Shared timed mutex types shall satisfy the requirements of timed mutex types (32.4.3.3), shared mutex types (32.4.3.4), and additionally shall satisfy 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).
2. The expression `m.try_lock_shared_for(rel_time)` shall be well-formed and have the following semantics:
   - **Requires**: 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_...`
The function shall return within the timeout specified by \texttt{rel\_time} only if it has obtained shared ownership of the mutex object. [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] If an exception is thrown then a shared lock shall not have been acquired for the current thread.

\textbf{Return type:} \texttt{bool}.

\textbf{Returns:} \texttt{true} if the shared lock was acquired, \texttt{false} otherwise.

\textbf{Synchronization:} If \texttt{try\_lock\_shared\_for()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (6.8.2) this operation.

\textbf{Throws:} Timeout-related exceptions (32.2.4).

The expression \texttt{m.try\_lock\_shared\_until(abs\_time)} shall be well-formed and have the following semantics:

\textbf{Requires:} The calling thread has no ownership of the mutex.

\textbf{Effects:} The function attempts to obtain shared ownership of the mutex. If \texttt{abs\_time} has already passed, the function attempts to obtain shared ownership without blocking (as if by calling \texttt{try\_lock\_shared()}). The function shall return before the absolute timeout (32.2.4) specified by \texttt{abs\_time} only if it has obtained shared ownership of the mutex object. [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] If an exception is thrown then a shared lock shall not have been acquired for the current thread.

\textbf{Return type:} \texttt{bool}.

\textbf{Returns:} \texttt{true} if the shared lock was acquired, \texttt{false} otherwise.

\textbf{Synchronization:} If \texttt{try\_lock\_shared\_until()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (6.8.2) this operation.

\textbf{Throws:} Timeout-related exceptions (32.2.4).

\subsection{Class \texttt{shared\_timed\_mutex}}

\begin{verbatim}
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 \texttt{shared\_timed\_mutex} provides a non-recursive mutex with shared ownership semantics.
\end{verbatim}
The class `shared_timed_mutex` shall satisfy all of the shared timed mutex requirements (32.4.3.5). It shall be a standard-layout class (11.1).

The behavior of a program is undefined if:

1. it destroys a `shared_timed_mutex` object owned by any thread,
2. a thread attempts to recursively gain any ownership of a `shared_timed_mutex`, or
3. a thread terminates while possessing any ownership of a `shared_timed_mutex`.

32.4.4 Locks

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]

Some lock constructors take tag types which describe what should be done with the lockable object during the lock’s construction.

```cpp
namespace std {
    struct defer_lock_t {}; // do not acquire ownership of the mutex
    struct try_to_lock_t {}; // try to acquire ownership of the mutex
    // without blocking
    struct adopt_lock_t {}; // assume the calling thread has already
    // obtained mutex ownership and manage it

    inline constexpr defer_lock_t defer_lock { };    // do not acquire ownership of the mutex
    inline constexpr try_to_lock_t try_to_lock { };  // try to acquire ownership of the mutex
    // without blocking
    inline constexpr adopt_lock_t adopt_lock { };   // assume the calling thread has already
    // obtained mutex ownership and manage it
}
```

32.4.4.1 Class template `lock_guard`

```cpp
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 (6.6.3). 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 satisfy the `Cpp17BasicLockable` requirements (32.2.5.2).

```cpp
explicit lock_guard(mutex_type& m);
```

Requires: 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);
```

Requires: The calling thread owns the mutex `m`.

Effects: Initializes `pm` with `m`.
6  **Throws:** Nothing.

~`lock_guard();`

7  **Effects:** As if by `pm.unlock()`.

### 32.4.4.2 Class template `scoped_lock` [thread.lock.scoped]

```cpp
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
};
```

1 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 (6.6.3). 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 satisfy the `Cpp17BasicLockable` requirements (32.2.5.2). Otherwise, each of the mutex types shall satisfy the `Cpp17Lockable` requirements (32.2.5.3).

```cpp
explicit scoped_lock(MutexTypes&... m);
```

2  **Requires:** If a `MutexTypes` type is not a recursive mutex, the calling thread does not own the corresponding mutex element of `m`.

3  **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  **Requires:** The calling thread owns all the mutexes in `m`.

5  **Effects:** Initializes `pm` with `tie(m...)`.

6  **Throws:** Nothing.

~`scoped_lock();`

7  **Effects:** For all `i` in `[0, sizeof...(MutexTypes))`, `get<i>(pm).unlock()`.

### 32.4.4.3 Class template `unique_lock` [thread.lock.unique]

```cpp
class unique_lock {

public:
    using mutex_type = Mutex;

    // 32.4.4.3.1, construct/copy/destroy
    unique_lock() noexcept;
    explicit unique_lock(mutex_type& m);
    template<class Mutex, 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);

// 32.4.4.3.2, 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();

// 32.4.4.3.3, modifiers
void swap(unique_lock& u) noexcept;
mutex_type* release() noexcept;

// 32.4.4.3.4, 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;

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 (6.6.3) of the unique_lock object. The supplied Mutex type shall satisfy the Cpp17BasicLockable requirements (32.2.5.2).

[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]

32.4.4.3.1  Constructors, destructor, and assignment

unique_lock() noexcept;

Effects: Constructs an object of type unique_lock.

Ensures: pm == 0 and owns == false.

explicit unique_lock(mutex_type& m);

Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex.

Effects: Constructs an object of type unique_lock and calls m.lock().

Ensures: pm == addressof(m) and owns == true.
unique_lock(mutex_type& m, defer_lock_t) noexcept;
Effects: Constructs an object of type unique_lock.
Ensures: pm == addressof(m) and owns == false.

unique_lock(mutex_type& m, try_to_lock_t);
Requires: The supplied Mutex type shall satisfy the Cpp17Lockable requirements (32.2.5.3). If mutex_type is not a recursive mutex the calling thread does not own the mutex.
Effects: Constructs an object of type unique_lock and calls m.try_lock().
Ensures: 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);
Requires: The calling thread owns the mutex.
Effects: Constructs an object of type unique_lock.
Ensures: 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);
Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The supplied Mutex type shall satisfy the Cpp17TimedLockable requirements (32.2.5.4).
Effects: Constructs an object of type unique_lock and calls m.try_lock_until(abs_time).
Ensures: 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);
Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The supplied Mutex type shall satisfy the Cpp17TimedLockable requirements (32.2.5.4).
Effects: Constructs an object of type unique_lock and calls m.try_lock_for(rel_time).
Ensures: 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;
Ensures: 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().
Ensures: 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().

32.4.4.3.2 Locking

void lock();
Effects: As if by pm->lock().
Ensures: owns == true.
Throws: Any exception thrown by `pm->lock()`. `system_error` when an exception is required (32.2.2).

Error conditions:

(4.1) operation_not_permitted — if `pm` is `nullptr`.
(4.2) resource_deadlock_would_occur — if on entry `owns` is true.

```cpp
bool try_lock();
```

**Requires:** The supplied Mutex shall satisfy the `Cpp17Lockable` requirements (32.2.5.3).

**Effects:** As if by `pm->try_lock()`.

**Returns:** The value returned by the call to `try_lock()`.

**Ensures:** `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:**

(10.1) operation_not_permitted — if `pm` is `nullptr`.
(10.2) resource_deadlock_would_occur — if on entry `owns` is true.

```cpp
template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

**Requires:** The supplied Mutex type shall satisfy 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)`.

**Ensures:** `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:**

(16.1) operation_not_permitted — if `pm` is `nullptr`.
(16.2) resource_deadlock_would_occur — if on entry `owns` is true.

```cpp
template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

**Requires:** The supplied Mutex type shall satisfy 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_until(rel_time)`.

**Ensures:** `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:**

(22.1) operation_not_permitted — if `pm` is `nullptr`.
(22.2) resource_deadlock_would_occur — if on entry `owns` is true.

```cpp
void unlock();
```

**Effects:** As if by `pm->unlock()`.

**Ensures:** `owns == false`.

**Throws:** `system_error` when an exception is required (32.2.2).

**Error conditions:**

(26.1) operation_not_permitted — if on entry `owns` is `false`. 

§ 32.4.4.3.2
32.4.4.3.3 Modifiers

```cpp
void swap(unique_lock& u) noexcept;
```

1 Effects: Swaps the data members of *this and u.

```cpp
mutex_type* release() noexcept;
```

2 Returns: The previous value of pm.
3 Ensures: pm == 0 and owns == false.

```cpp
template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```

4 Effects: As if by x.swap(y).

32.4.4.3.4 Observers

```cpp
bool owns_lock() const noexcept;
```

1 Returns: owns.

```cpp
explicit operator bool() const noexcept;
```

2 Returns: owns.

```cpp
mutex_type *mutex() const noexcept;
```

3 Returns: pm.

32.4.4.4 Class template shared_lock

```cpp
namespace std {
    template<class Mutex>
    class shared_lock {
        public:
            using mutex_type = Mutex;

            // 32.4.4.4.1, 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;

            // 32.4.4.4.2, 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();

            // 32.4.4.4.3, modifiers
            void swap(shared_lock& u) noexcept;
            mutex_type* release() noexcept;
        }
    } // namespace std
```
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 (6.6.3) of the `shared_lock` object. The supplied `Mutex` type shall satisfy the shared mutex requirements (32.4.3.5).

[Note: `shared_lock<Mutex>` meets the `Cpp17TimedLockable` requirements (32.2.5.4). — end note]

### 32.4.4.4.1 Constructors, destructor, and assignment

```cpp
shared_lock() noexcept;
```

**Effects:** Constructs an object of type `shared_lock`.

**Ensures:** `pm == nullptr` and `owns == false`.

```cpp
explicit shared_lock(mutex_type& m);
```

**Requires:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Constructs an object of type `shared_lock` and calls `m.lock_shared()`.

**Ensures:** `pm == addressof(m)` and `owns == true`.

```cpp
shared_lock(mutex_type& m, defer_lock_t) noexcept;
```

**Effects:** Constructs an object of type `shared_lock`.

**Ensures:** `pm == addressof(m)` and `owns == false`.

```cpp
shared_lock(mutex_type& m, try_to_lock_t);
```

**Requires:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Constructs an object of type `shared_lock` and calls `m.try_lock_shared()`.

**Ensures:** `pm == addressof(m)` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared()`.

```cpp
shared_lock(mutex_type& m, adopt_lock_t);
```

**Requires:** The calling thread has shared ownership of the mutex.

**Effects:** Constructs an object of type `shared_lock`.

**Ensures:** `pm == addressof(m)` and `owns == true`.

```cpp
template<class Clock, class Duration>
shared_lock(mutex_type& m,
    const chrono::time_point<Clock, Duration>& abs_time);
```

**Requires:** The calling thread does not own the mutex for any ownership mode.

**Effects:** Constructs an object of type `shared_lock` and calls `m.try_lock_shared_until(abs_time)`.

**Ensures:** `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);

17 Requires: The calling thread does not own the mutex for any ownership mode.

18 Effects: Constructs an object of type shared_lock and calls m.try_lock_shared_for(rel_time).

19 Ensures: 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();

20 Effects: If owns calls pm->unlock_shared().

shared_lock(shared_lock&& sl) noexcept;

21 Ensures: 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;

22 Effects: If owns calls pm->unlock_shared().

23 Ensures: 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.

32.4.4.4.2 Locking

void lock();

1 Effects: As if by pm->lock_shared().

2 Ensures: owns == true.

3 Throws: Any exception thrown by pm->lock_shared(). system_error when an exception is required (32.2.2).

4 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();

5 Effects: As if by pm->try_lock_shared().

6 Returns: The value returned by the call to pm->try_lock_shared().

7 Ensures: owns == res, where res is the value returned by the call to pm->try_lock_shared().

8 Throws: Any exception thrown by pm->try_lock_shared(). system_error when an exception is required (32.2.2).

9 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);

10 Effects: As if by pm->try_lock_shared_until(abs_time).

11 Returns: The value returned by the call to pm->try_lock_shared_until(abs_time).

12 Ensures: owns == res, where res is the value returned by the call to pm->try_lock_shared_until(abs_time).

13 Throws: Any exception thrown by pm->try_lock_shared_until(abs_time). system_error when an exception is required (32.2.2).

14 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).
Ensures: 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:
— operation_not_permitted — if pm is nullptr.
— resource_deadlock_would_occur — if on entry owns is true.

void unlock();
Effects: As if by pm->unlock_shared().
Ensures: owns == false.
Throws: system_error when an exception is required (32.2.2).

Error conditions:
— operation_not_permitted — if on entry owns is false.

32.4.4.4.3 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.
Ensures: pm == nullptr and owns == false.

32.4.4.4.4 Observers

bool owns_lock() const noexcept;
Returns: owns.

explicit operator bool() const noexcept;
Returns: owns.

mutex_type* mutex() const noexcept;
Returns: pm.

32.4.5 Generic locking algorithms

template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
Requires: Each template parameter type shall satisfy the Cpp17Lockable requirements. [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]
Effects: Calls try_lock() for each argument in order beginning with the first until all arguments have been processed or a call to try_lock() fails, either by returning false or by throwing an exception. If a call to try_lock() fails, unlock() is called for all prior arguments with no further calls to try_lock().
Returns: -1 if all calls to try_lock() returned true, otherwise a zero-based index value that indicates the argument for which try_lock() returned false.
template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

Requires: Each template parameter type shall satisfy the Cpp17Lockable requirements, [Note: The unique_lock class template meets these requirements when suitably instantiated. — end note]

Effects: All arguments are locked via a sequence of calls to lock(), try_lock(), or unlock() on each argument. The sequence of calls does not result in deadlock, but is otherwise unspecified. [Note: 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().

32.4.6 Call once

32.4.6.1 Struct once_flag

namespace std {
    struct once_flag {
        constexpr once_flag() noexcept;
        once_flag(const once_flag&) = delete;
        once_flag& operator=(const once_flag&) = delete;
    };
}

The class once_flag is an opaque data structure that call_once uses to initialize data without causing a data race or deadlock.

call_once

namespace std {
    struct once_flag {
        constexpr once_flag() noexcept;
        once_flag(const once_flag&) = delete;
        once_flag& operator=(const once_flag&) = delete;
    };
}

The class once_flag is an opaque data structure that call_once uses to initialize data without causing a data race or deadlock.

call_once

32.4.6.2 Function call_once

template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& func, Args&&... args);

Requires: INVOKE(std::forward<Callable>(func), std::forward<Args>(args)...)
(see 20.14.3) shall be a valid expression.

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 shall call 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 shall propagate the exception to the caller of call_once. Among all executions of call_once for any given once_flag: at most one shall be a returning execution; if there is a returning execution, it shall be 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]

Synchronization: For any given once_flag: all active executions occur in a total order; completion of an active execution synchronizes with (6.8.2) the start of the next one in this total order; and the returning execution synchronizes with the return from all passive executions.

Throws: system_error when an exception is required (32.2.2), or any exception thrown by func.

[Example:]
// global flag, regular function
void init();
std::once_flag flag;
32.5 Condition variables

1 Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class `condition_variable` provides a condition variable that can only wait on an object of type `unique_lock<mutex>`, allowing the implementation to be more efficient. Class `condition_variable_any` provides a general condition variable that can wait on objects of user-supplied lock types.

2 Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

3 The execution of `notify_one` and `notify_all` shall be atomic. The execution of `wait`, `wait_for`, and `wait_until` shall be performed in three atomic parts:
   1. the release of the mutex and entry into the waiting state;
   2. the unblocking of the wait; and
   3. the reacquisition of the lock.

4 The implementation shall behave as if all executions of `notify_one`, `notify_all`, and each part of the `wait`, `wait_for`, and `wait_until` executions are executed in a single unspecified total order consistent with the 'happens before' order.

5 Condition variable construction and destruction need not be synchronized.

32.5.1 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 };
}
```

32.5.2 Non-member functions

```cpp
void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);
```

1 Requires: `lk` is locked by the calling thread and either

---

© ISO/IEC 2023

---

§ 32.5.2 1525
— \texttt{lk.mutex()} returns the same value for each of the lock arguments supplied by all concurrently
waiting (via \texttt{wait}, \texttt{wait\_for}, or \texttt{wait\_until}) threads.

2 \hspace{1em} \textbf{Effects:} Transfers ownership of the lock associated with \texttt{lk} into internal storage and schedules \texttt{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 shall be as if:

\begin{verbatim}
lk.unlock();
cond.notify\_all();
\end{verbatim}

3 \hspace{1em} \textbf{Synchronization:} The implied \texttt{lk.unlock()} call is sequenced after the destruction of all objects with
thread storage duration associated with the current thread.

4 \hspace{1em} [\textbf{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 \texttt{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 \hspace{1em} [\textbf{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 \texttt{lk}, and that this lock is not released and reacquired
prior to calling \texttt{notify\_all\_at\_thread\_exit}. — end note]

32.5.3 \textbf{Class} condition\_variable \hspace{1em} [thread.condition.condvar]

\begin{verbatim}
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; \hspace{1em} // see 32.2.3
            native\_handle\_type native\_handle(); \hspace{1em} // see 32.2.3
        }
    }
\end{verbatim}

1 \hspace{1em} The class \texttt{condition\_variable} shall be a standard-layout class (11.1).

\begin{verbatim}
    condition\_variable();
\end{verbatim}

2 \hspace{1em} \textbf{Effects:} Constructs an object of type \texttt{condition\_variable}.

3 \hspace{1em} \textbf{Throws:} \texttt{system\_error} when an exception is required (32.2.2).

4 \hspace{1em} \textbf{Error conditions:}
— resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

~condition_variable();

5  Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they may 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]

6  Effects: Destroys the object.

void notify_one() noexcept;

7  Effects: If any threads are blocked waiting for *this, unblocks one of those threads.

void notify_all() noexcept;

8  Effects: Unblocks all threads that are blocked waiting for *this.

void wait(unique_lock<mutex>& lock);

9  Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(9.1) — no other thread is waiting on this condition_variable object or
(9.2) — 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.

10 Effects:
(10.1) — Atomically calls lock.unlock() and blocks on *this.
(10.2) — When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
(10.3) — The function will unblock when signaled by a call to notify_one() or a call to notify_all(), or spuriously.

11 Remarks: If the function fails to meet the postcondition, terminate() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

12 Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

13 Throws: Nothing.

template<class Predicate>
void wait(unique_lock<mutex>& lock, Predicate pred);

14 Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(14.1) — no other thread is waiting on this condition_variable object or
(14.2) — 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.

15 Effects: Equivalent to:
while (!pred())
  wait(lock);

16 Remarks: If the function fails to meet the postcondition, terminate() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

17 Ensures: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

18 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);

19 Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(19.1) — no other thread is waiting on this condition_variable object or
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

**Effects:**

— Atomically calls lock.unlock() and blocks on *this.
— When unblocked, calls lock.lock() (possibly blocking on the lock), then returns.
— The function will unblock when signaled by a call to notify_one(), a call to notify_all(), expiration of the absolute timeout (32.2.4) specified by abs_time, or spuriously.
— If the function exits via an exception, lock.lock() shall be called prior to exiting the function.

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Ensures:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

**Returns:** cv_status::timeout if the absolute timeout (32.2.4) specified by abs_time expired, otherwise cv_status::no_timeout.

**Throws:** Timeout-related exceptions (32.2.4).

```cpp
template<class Rep, class Period>
cv_status wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time);
```

**Requires:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

**Effects:** Equivalent to:

```cpp
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

**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() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Ensures:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

**Throws:** Timeout-related exceptions (32.2.4).

```cpp
template<class Clock, class Duration, class Predicate>
bool wait_until(unique_lock<mutex>& lock,
    const chrono::time_point<Clock, Duration>& abs_time,
    Predicate pred);
```

**Requires:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either

— no other thread is waiting on this condition_variable object or
— lock.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

**Effects:** Equivalent to:

```cpp
while (!pred())
    if (wait_until(lock, abs_time) == cv_status::timeout)
        return pred();
return true;
```

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Ensures:** 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.

§ 32.5.3
template<class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time,
    Predicate pred);

Requires: lock.owns_lock() is true and lock.mutex() is locked by the calling thread, and either
(37.1) — no other thread is waiting on this condition_variable object or
(37.2) — 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() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: 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.

32.5.4 Class condition_variable_any [thread.condition.condvarany]

A Lock type shall satisfy 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;
        template<class Lock>
            void wait(Lock& lock);
        template<class Lock, class Predicate>
            void wait(Lock& lock, Predicate pred);

        template<class Lock, class Clock, class Duration>
            cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
        template<class Lock, class Clock, class Duration, class Predicate>
            bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time,
                Predicate pred);
        template<class Lock, class Rep, class Period>
            cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
        template<class Lock, class Rep, class Period, class Predicate>
            bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
    }
}

condition_variable_any();

Effects: Constructs an object of type condition_variable_any.

Throws: bad_alloc or system_error when an exception is required (32.2.2).
Error conditions:

(4.1) - resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

(4.2) - operation_not_permitted — if the thread does not have the privilege to perform the operation.

~condition_variable_any();

Requires: There shall be no thread blocked on *this. [Note: That is, all threads shall have been notified; they may 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]

Effects: Destroys the object.

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.

template<class Lock>
void wait(Lock& lock);

Effects:

(9.1) — Atomically calls lock.unlock() and blocks on *this.

(9.2) — When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.

(9.3) — The function will unblock when signaled by a call to notify_one(), a call to notify_all(), or spuriously.

Remarks: If the function fails to meet the postcondition, terminate() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: lock is locked by the calling thread.

Throws: Nothing.

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:

(14.1) — Atomically calls lock.unlock() and blocks on *this.

(14.2) — When unblocked, calls lock.lock() (possibly blocking on the lock) and returns.

(14.3) — 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.

(14.4) — If the function exits via an exception, lock.lock() shall be called prior to exiting the function.

Remarks: If the function fails to meet the postcondition, terminate() shall be called (14.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: 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).
```cpp
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() shall be called (14.5.1). [Note: This
    can happen if the re-locking of the mutex throws an exception. — end note]

Ensures: 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 Futures

#### 32.6.1 Overview

32.6 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]

#### 32.6.2 Header `<future>` synopsis

```cpp
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 R> class packaged_task; // not defined

template<class R, class... ArgTypes> 
class packaged_task<R(ArgTypes...)>
{ }

template<class R, class... ArgTypes> 
void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

template<class F, class... Args> 
[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(F&& f, Args&&... args);

template<class F, class... Args> 
[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(launch policy, F&& f, Args&&... args);

1 The enum type launch is a bitmask type (16.4.2.2.4) with elements launch::async and launch::deferred.
[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.

32.6.3 Error handling

const error_category& future_category() noexcept;

Returns: A reference to an object of a type derived from class error_category.

1 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
"future".

error_code make_error_code(future_errc e) noexcept;

Returns: error_code(static_cast<int>(e), future_category()).

2 error_condition make_error_condition(future_errc e) noexcept;

Returns: error_condition(static_cast<int>(e), future_category()).
### 32.6.4 Class `future_error` [futures.future.error]

```
namespace std {
    class future_error : public logic_error {
        public:
            explicit future_error(future_errc e);

            const error_code& code() const noexcept;
            const char* what() const noexcept;

        private:
            error_code ec_;          // exposition only
        }
    }
```

**Effects:** Constructs an object of class `future_error` and initializes `ec_` with `make_error_code(e)`.

```
const error_code& code() const noexcept;

Returns: `ec_`.
```

```
const char* what() const noexcept;

Returns: An NTBS incorporating `code().message()`.
```

### 32.6.5 Shared state [futures.state]

1 Many of the classes introduced in this subclause 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*]

3 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:

(5.1) — if the return object or provider holds the last reference to its shared state, the shared state is destroyed; and

(5.2) — the return object or provider gives up its reference to its shared state; and

(5.3) — these actions will not block for the shared state to become ready, except that it may block if all of the following are true: the shared state was created by a call to `std::async`, the shared state is not yet ready, and this was the last reference to the shared state.

6 When an asynchronous provider is said to make its shared state ready, it means:

(6.1) — first, the provider marks its shared state as ready; and

(6.2) — second, the provider unblocks any execution agents waiting for its shared state to become ready.

7 When an asynchronous provider is said to abandon its shared state, it means:

(7.1) — first, if that state is not ready, the provider

(7.1.1) — stores an exception object of type `future_error` with an error condition of `broken_promise` within its shared state; and then

(7.1.2) — makes its shared state ready;

(7.2) — second, the provider releases its shared state.
A shared state is ready only if it holds a value or an exception ready for retrieval. Waiting for a shared state to become ready may invoke code to compute the result on the waiting thread if so specified in the description of the class or function that creates the state object.

Calls to functions that successfully set the stored result of a shared state synchronize with (6.8.2) 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 (6.8.2) 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 (6.6.5.2) is sequenced before making that shared state ready.

Access to the result of the same shared state may conflict (6.8.2). [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 (16.5.5.9). For example, concurrent accesses through references returned by `shared_future::get()` (32.6.8) must either use read-only operations or provide additional synchronization. — end note]

### 32.6.6 Class template promise

```cpp
namespace std {
    template<class R>
    class promise {
    public:
        promise();
        template<class Allocator>
        promise(allocator_arg_t, const Allocator& a);
        promise(promise&& rhs) noexcept;
        promise(const promise& rhs) = delete;
        ~promise();

        // assignment
        promise& operator=(promise&& rhs) noexcept;
        promise& operator=(const promise& rhs) = delete;
        void swap(promise& other) noexcept;

        // retrieving the result
        future<R> get_future();

        // setting the result
        void set_value(see below);
        void set_exception(exception_ptr p);

        // setting the result with deferred notification
        void set_value_at_thread_exit(see below);
        void set_exception_at_thread_exit(exception_ptr p);
    }

    template<class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;

    template<class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>;
}
```

The implementation shall provide 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.
promise();

3

Requires: Alloc shall satisfy the Cpp17Allocator requirements (Table 34).

template<class Allocator>
    promise(allocator_arg_t, const Allocator& a);

4

Effects: Constructs a promise object and a shared state. The second constructor uses the allocator a to allocate memory for the shared state.

promise(promise&& rhs) noexcept;

5

Effects: Constructs a new promise object and transfers ownership of the shared state of rhs (if any) to the newly-constructed object.

Ensures: rhs has no shared state.

~promise();

6

Effects: Abandons any shared state (32.6.5).

promise& operator=(promise&& rhs) noexcept;

7

Effects: Abandons any shared state (32.6.5) and then as if promise(std::move(rhs)).swap(*this).

Returns: *this.

void swap(promise& other) noexcept;

8

Effects: Exchanges the shared state of *this and other.

Ensures: *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

future<R> get_future();

9

Returns: A future<R> object with the same shared state as *this.

Synchronization: Calls to this function do not introduce data races (6.8.2) 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]

Throws: future_error if *this has no shared state or if get_future has already been called on a promise with the same shared state as *this.

Error conditions:

(15.1) future_already_retrieved if get_future has already been called on a promise with the same shared state as *this.

(15.2) no_state if *this has no shared state.

void promise::set_value(const R& r);

10

void promise::set_value(R&& r);

void promise<R&>::set_value(R& r);

void promise<void>::set_value();

11

Effects: Atomically stores the value r in the shared state and makes that state ready (32.6.5).

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);

12

Requires: p is not null.
Effects: Atomically stores the exception pointer p in the shared state and makes that state ready (32.6.5).

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);

Requires: 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).

32.6.7 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 (32.6.5) 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]

namespace std {
    template<class R>
    class future {

    § 32.6.7 1536
public:
    future() noexcept;
    future(future&&) noexcept;
    future(const future& rhs) = delete;
    ~future();
    future& operator=(const future& rhs) = 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 shall provide 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;
5  Effects: Constructs an empty future object that does not refer to a shared state.
6  Ensures: valid() == false.

future(future&& rhs) noexcept;
7  Effects: Move constructs a future object that refers to the shared state that was originally referred to by rhs (if any).
8  Ensures:
(8.1) valid() returns the same value as rhs.valid() prior to the constructor invocation.
(8.2) rhs.valid() == false.

~future();
9  Effects:
(9.1) Releases any shared state (32.6.5);
(9.2) destroys *this.

future& operator=(future&& rhs) noexcept;
10 Effects:
(10.1) Releases any shared state (32.6.5).
(10.2) move assigns the contents of rhs to *this.
11 Ensures:
(11.1) valid() returns the same value as rhs.valid() prior to the assignment.
(11.2) rhs.valid() == false.

shared_future<R> share() noexcept;
12 Returns: shared_future<R>(std::move(*this)).
13 Ensures: valid() == false.
void future<void>::get();

[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]

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 (32.6.5).

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.

Ensures: 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 (32.6.9), 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 (32.6.9), otherwise blocks until the shared state is ready or until the absolute timeout (32.2.4) specified by abs_time has expired.

Returns:

(25.1) — future_status::deferred if the shared state contains a deferred function.
(25.2) — future_status::ready if the shared state is ready.
(25.3) — future_status::timeout if the function is returning because the absolute timeout (32.2.4) specified by abs_time has expired.

Throws: timeout-related exceptions (32.2.4).

32.6.8 Class template shared_future

The class template shared_future defines a type for asynchronous return objects which may share their shared state with other asynchronous return objects. A default-constructed shared_future object has no shared state. A shared_future object with shared state can be created by conversion from a future object and shares its shared state with the original asynchronous provider (32.6.5) of the shared state. The result (value or exception) of a shared_future object can be set by calling a respective function on an object that shares the same shared state.

[Note: 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]

namespace std {
    template<class R>
    class shared_future {
        public:
            shared_future() noexcept;
            shared_future(const shared_future& rhs) noexcept;
            shared_future(future<R>&& rhs) 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;
    }
}

4 The implementation shall provide 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;
5 
    Effects: Constructs an empty shared_future object that does not refer to a shared state.

    Ensures: valid() == false.

shared_future(const shared_future& rhs) noexcept;
6 
    Effects: Constructs a shared_future object that refers to the same shared state as rhs (if any).

    Ensures: valid() returns the same value as rhs.valid().

shared_future(future<R>&& rhs) noexcept;
shared_future(shared_future&& rhs) noexcept;
7 
    Effects: Move constructs a shared_future object that refers to the shared state that was originally referred to by rhs (if any).

    Ensures:
    (10.1) valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
    (10.2) rhs.valid() == false.

~shared_future();
8 
    Effects:
    (11.1) Releases any shared state (32.6.5);
    (11.2) destroys *this.

shared_future& operator=(shared_future&& rhs) noexcept;
9 
    Effects:
© ISO/IEC

---

(12.1) Releases any shared state (32.6.5);

(12.2) move assigns the contents of \( \text{rhs} \) to \(*\text{this}\).

**Ensures:**

(13.1) \( \text{valid()} \) returns the same value as \( \text{rhs.valid()} \) returned prior to the assignment.

(13.2) \( \text{rhs.valid()} == \text{false} \).

\[
\text{shared_future\& operator=}(\text{const shared_future\&} \ \text{rhs}) \ \text{noexcept};
\]

**Effects:**

(14.1) Releases any shared state (32.6.5);

(14.2) assigns the contents of \( \text{rhs} \) to \(*\text{this}\). [Note: As a result, \(*\text{this}\) refers to the same shared state as \( \text{rhs} \) (if any). — end note]

**Ensures:** \( \text{valid()} == \text{rhs.valid()} \).

\[
\text{const R\& shared_future::get()} \ \text{const};
\]

\[
\text{R\& shared_future<R\&>::get()} \ \text{const};
\]

\[
\text{void shared_future<void>::get()} \ \text{const};
\]

[Note: As described above, the template and its two required specializations differ only in the return type and return value of the member function \text{get}. — end note]

[Note: Access to a value object stored in the shared state is unsynchronized, so programmers should apply only those operations on \( \text{R} \) that do not introduce a data race (6.8.2). — end note]

**Effects:** \text{wait()} until the shared state is ready, then retrieves the value stored in the shared state.

**Returns:**

(19.1) \( \text{shared_future::get()} \) 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 \text{shared_future} object that returned the reference. — end note]

(19.2) \( \text{shared_future<R\&>::get()} \) returns the reference stored as value in the object’s shared state.

(19.3) \( \text{shared_future<void>::get()} \) returns nothing.

**Throws:** The stored exception, if an exception was stored in the shared state.

\[
\text{bool valid()} \ \text{const noexcept};
\]

**Returns:** \text{true} only if \(*\text{this}\) refers to a shared state.

\[
\text{void wait()} \ \text{const};
\]

**Effects:** Blocks until the shared state is ready.

\[
\text{template<class Rep, class Period>}
\]

\[
\text{future_status wait_for(const chrono::duration<Rep, Period>&} \ \text{rel_time}) \ \text{const};
\]

**Effects:** None if the shared state contains a deferred function (32.6.9), otherwise blocks until the shared state is ready or until the relative timeout (32.2.4) specified by \text{rel_time} has expired.

**Returns:**

(24.1) \( \text{future_status::deferred} \) if the shared state contains a deferred function.

(24.2) \( \text{future_status::ready} \) if the shared state is ready.

(24.3) \( \text{future_status::timeout} \) if the function is returning because the relative timeout (32.2.4) specified by \text{rel_time} has expired.

**Throws:** timeout-related exceptions (32.2.4).

\[
\text{template<class Clock, class Duration>}
\]

\[
\text{future_status wait_until(const chrono::time_point<Clock, Duration>&} \ \text{abs_time}) \ \text{const};
\]

**Effects:** None if the shared state contains a deferred function (32.6.9), otherwise blocks until the shared state is ready or until the absolute timeout (32.2.4) specified by \text{abs_time} has expired.

**Returns:**

§ 32.6.8 1540
— 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.

28  Throws: timeout-related exceptions (32.2.4).

32.6.9 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.

\[
\text{template<class } F, \text{ class... Args}> \\
\text{[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>>...>>} \\
\text{async(F&& f, Args&&... args);} \\
\text{template<class } F, \text{ class... Args}> \\
\text{[[nodiscard]] future<invoke_result_t<decay_t<F>, decay_t<Args>>...>>} \\
\text{async(launch policy, F&& f, Args&&... args);} \\
\]

\[
\text{Requires: } F \text{ and each } T_i \text{ in Args shall satisfy the Cpp17MoveConstructible requirements, and} \\
\text{INVOKE(}\text{decay-copy(std::forward(F)(f))}, \\
\text{decay-copy(std::forward<Args>(args))...} \text{ // see 20.14.3, 32.3.2.2} \\
\]

shall be a valid expression.

\[
\text{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 \text{ 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):} \\
\text{(3.1) If launch::async is set in policy, calls INVOKE(}\text{decay-copy(std::forward(F)(f))}, \\
\text{decay-copy(std::forward<Args>(args))...} \ (20.14.3, 32.3.2.2) 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(}\text{decay-copy(std::forward(F)(f))}, \\
\text{decay-copy(std::forward<Args>(args))...} \text{ 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.} \\
\text{(3.2) 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(}\text{std::move(g)}, \\
\text{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 (32.6.5) on an asynchronous return object referring to this shared state shall invoke the deferred function in the thread that called the waiting function. Once evaluation of INVOKE(}\text{std::move(g)}, \\
\text{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] \\
\text{(3.3) 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 may block for the shared state to become ready. — end note]

Synchronization: Regardless of the provided policy argument,
the invocation of `async` synchronizes with (6.8.2) the invocation of `f`. [Note: This statement applies even when the corresponding `future` object is moved to another thread. — end note] ; and

the completion of the function `f` is sequenced before (6.8.2) 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,

— 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 (32.3.2.5);

— the associated thread completion synchronizes with (6.8.2) 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.

6 **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.

7 **Error conditions:**

— `resource_unavailable_try_again` — if `policy == launch::async` and the system is unable to start a new thread.

8 **Example:**

```
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]

32.6.10 *Class template* `packaged_task` **[futures.task]**

The class template `packaged_task` defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.

When the `packaged_task` object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the shared state. Any futures that share the shared state will then be able to access the stored result.

```cpp
namespace std {
  template<class> class packaged_task;  // not defined

  template<class R, class... ArgTypes>
  class packaged_task<R(ArgTypes...)>
  {
    public:
    // construction and destruction
    packaged_task() noexcept;
    template<class F>
    explicit packaged_task(F&& f);
    ~packaged_task();

    // no copy
    packaged_task(const packaged_task&) = delete;
    packaged_task& operator=(const packaged_task&) = delete;

    // move support
    packaged_task(packaged_task&& rhs) noexcept;
    packaged_task& operator=(packaged_task&& rhs) noexcept;
    void swap(packaged_task other) noexcept;

    bool valid() const noexcept;

  } // class packaged_task
```
// 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;

32.6.10.1 Member functions

packaged_task() noexcept;

Effects: Constructs a packaged_task object with no shared state and no stored task.

template<class F>
packaged_task(F&& f);

Requires: INVOKE<R>({f, t₁, t₂, ..., tₙ}) (20.14.3), where t₁, t₂, ..., tₙ are values of the corresponding types in ArgTypes..., shall be a valid expression. Invoking a copy of f shall behave the same as invoking f.

Remarks: This constructor shall not participate in overload resolution if remove_cvref_t<F> is the same type as packaged_task<R(ArgTypes...)>

Effects: Constructs a new packaged_task object with a shared state and initializes the object’s stored task with std::forward<F>(f).

Throws: Any exceptions thrown by the copy or move constructor of f, or bad_alloc if memory for the internal data structures could not be allocated.

packaged_task(packaged_task&& rhs) noexcept;

Effects: Constructs a new packaged_task object and transfers ownership of rhs’s shared state to *this, leaving rhs with no shared state. Moves the stored task from rhs to *this.

Ensures: rhs has no shared state.

packaged_task& operator=(packaged_task&& rhs) noexcept;

Effects:

— Releases any shared state (32.6.5);

— calls packaged_task(std::move(rhs)).swap(*this).

~packaged_task();

Effects: Abandons any shared state (32.6.5).

void swap(packaged_task& other) noexcept;

Effects: Exchanges the shared states and stored tasks of *this and other.

Ensures: *this has the same shared state and stored task (if any) as other prior to the call to swap. other has the same shared state and stored task (if any) as *this prior to the call to swap.

bool valid() const noexcept;

Returns: true only if *this has a shared state.

future<R> get_future();

Returns: A future object that shares the same shared state as *this.

Synchronization: Calls to this function do not introduce data races (6.8.2) with calls to operator() or make_ready_at_thread_exit. [Note: Such calls need not synchronize with each other. — end note]

Throws: A future_error object if an error occurs.
Error conditions:

(16.1) future_already_retrieved if `get_future` has already been called on a `packaged_task` object with the same shared state as `*this`.

(16.2) no_state if `*this` has no shared state.

```cpp
void operator()(ArgTypes... args);
```

Effects: As if by `INVOKE<R>(f, t₁, t₂, ..., tₙ)` (20.14.3), where `f` is the stored task of `*this` and `t₁, t₂, ..., tₙ` are the values in `args...`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. The shared state of `*this` is made ready, and any threads blocked in a function waiting for the shared state of `*this` to become ready are unblocked.

Throws: A `future_error` exception object if there is no shared state or the stored task has already been invoked.

Error conditions:

(19.1) promise_already_satisfied if the stored task has already been invoked.

(19.2) no_state if `*this` has no shared state.

```cpp
void make_ready_at_thread_exit(ArgTypes... args);
```

Effects: As if by `INVOKE<R>(f, t₁, t₂, ..., tₙ)` (20.14.3), where `f` is the stored task and `t₁, t₂, ..., tₙ` are the values in `args...`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. In either case, this shall be done without making that state ready (32.6.5) immediately. Schedules the shared state to be made ready when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed.

Throws: `future_error` if an error condition occurs.

Error conditions:

(22.1) promise_already_satisfied if the stored task has already been invoked.

(22.2) 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 (32.6.5). —end note]

Throws:

(24.1) bad_alloc if memory for the new shared state could not be allocated.

(24.2) any exception thrown by the move constructor of the task stored in the shared state.

(24.3) future_error with an error condition of no_state if `*this` has no shared state.

32.6.10.2 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 (8.8, 9.1, 11.7) 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` (9.1.3), `namespace` (9.7.1), `class` (Clause 11), `enumeration` (9.6), and `template` (Clause 13) declarations.

```plaintext
typedef-name:
  identifier
  simple-template-id

namespace-name:
  identifier
  namespace-alias

namespace-alias:
  identifier

class-name:
  identifier
  simple-template-id

enum-name:
  identifier

template-name:
  identifier
```

A.2 Lexical conventions

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

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

preprocessing-token:
  header-name
  identifier
  pp-number
  character-literal
  user-defined-character-literal
  string-literal
  user-defined-string-literal
  preprocessing-op-or-punc

  each non-white-space character that cannot be one of the above

token:
  identifier
  keyword
  literal
  operator
  punctuator

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

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

literal:
  integer-literal
  character-literal
  floating-literal
  string-literal
  boolean-literal
  pointer-literal
  user-defined-literal

integer-literal:
  binary-literal integer-suffixopt
  octal-literal integer-suffixopt
  decimal-literal integer-suffixopt
  hexadecimal-literal integer-suffixopt
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
hexadecimal-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

ingeger-suffix:
  unsigned-suffix long-suffix
  unsigned-suffix long-long-suffix
  long-suffix unsigned-suffix
  long-long-suffix unsigned-suffix

unsigned-suffix:
  one of
  u U
long-suffix:
  one of
  l L
long-long-suffix:
  one of
  ll LL

character-literal:
  encoding-prefix \opt c-char-sequence
encoding-prefix:
  one of
  u8 u U L

\opt c-char-sequence

\opt c-char
\opt c-char-sequence c-char

c-char:
  any member of the source character set except the single-quote \', backslash \\, or new-line character
  escape-sequence
  universal-character-name

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

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

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

defloating-literal:
  decimal-floating-literal
  hexadecimal-floating-literal
decimal-floating-literal:
  fractional-constant exponent-part_opt floating-suffix_opt
digit-sequence exponent-part floating-suffix_opt
defloating-literal:
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffix_opt
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-suffix_opt
defractional-constant:
  digit-sequence_opt . digit-sequence
digit-sequence .

hexadecimal-floating-literal:
  hexadecimal-digit-sequence_opt . hexadecimal-digit-sequence
  hexadecimal-digit-sequence .

eponent-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 , digit_opt

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

boolean-literal:
    false
    true

pointer-literal:
    nullptr

user-defined-literal:
    user-defined-integer-literal
    user-defined-floating-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-literal:
    fractional-constant exponent-part_opt ud-suffix
    digit-sequence exponent-part ud-suffix
    hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part ud-suffix
    hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part ud-suffix

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

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

ud-suffix:
    identifier

A.3 Basics

translation-unit:
    top-level-declaration-seq_opt
    global-module-fragment_opt module-declaration top-level-declaration-seq_opt private-module-fragment_opt

private-module-fragment:
    module : private ; top-level-declaration-seq_opt

top-level-declaration-seq:
    top-level-declaration
    top-level-declaration-seq top-level-declaration

top-level-declaration:
    module-import-declaration
    declaration

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 compound-statement
  lambda-introducer lambda-declarator requires-clause_opt compound-statement
  lambda-introducer < template-parameter-list > requires-clause_opt compound-statement
  lambda-introducer < template-parameter-list > requires-clause_opt
  lambda-declarator requires-clause_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

lambda-capture:
  capture-default
  capture-list
  capture-default , capture-list

capture-default:
  &
  =

capture-list:
  capture
  capture-list , capture

capture:
  simple-capture ... opt
  ... opt init-capture

simple-capture:
  identifier
  & identifier
  this
  * this

init-capture:
  identifier initializer
  & 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:
  trailing-return-type
  \rightarrow\ 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 \rightarrow 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

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
nenptr-new-declarator

noptr-new-declarator:
   [ expression_list opt ] attribute-specifier-seq opt
nenptr-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

noexcept-expression:
   noexcept ( 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

compare-expression:
   shift-expression
   compare-expression <=> shift-expression

relational-expression:
   compare-expression
   relational-expression < compare-expression
   relational-expression > compare-expression
   relational-expression <= compare-expression
   relational-expression >= compare-expression
equality-expression:
  relational-expression
  equality-expression == relational-expression
  equality-expression != relational-expression

and-expression:
  equality-expression
  and-expression & equality-expression

exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ and-expression

inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression

logical-and-expression:
  inclusive-or-expression
  logical-and-expression && inclusive-or-expression

logical-or-expression:
  logical-and-expression
  logical-or-expression || logical-and-expression

classical-expression:
  logical-or-expression
  logical-or-expression ?: assignment-expression

yield-expression:
  co_yield assignment-expression
  co_yield braced-init-list

throw-expression:
  throw assignment-expression

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  yield-expression
  throw-expression

assignment-operator:
  one of
  = *= /= %= += -= >>= <<= &= ^= |=

classical-expression:
  conditional-expression

constant-expression:
  conditional-expression

A.5 Statements

statement:
  labeled-statement
  attribute-specifier-seq expression-statement
  attribute-specifier-seq compound-statement
  attribute-specifier-seq selection-statement
  attribute-specifier-seq iteration-statement
  attribute-specifier-seq jump-statement
  declaration-statement
  attribute-specifier-seq try-block

init-statement:
  expression-statement
  simple-declaration

condition:
  expression
  attribute-specifier-seq decl-specifier-seq declarator brace-or-equal-initializer
labeled-statement:
  attribute-specifier-seqopt identifier : statement
  attribute-specifier-seqopt case constant-expression : statement
  attribute-specifier-seqopt default : statement

expression-statement:
  expressionopt ;

compound-statement:
  { statement-seqopt }

statement-seq:
  statement
  statement-seq statement

selection-statement:
  if constexpr opt ( init-statementopt condition ) statement
  if constexpr opt ( init-statementopt condition ) statement else statement
  switch ( init-statementopt condition ) statement

iteration-statement:
  while ( condition ) statement
  do statement while ( expression ) ;
  for ( init-statement conditionopt ; expressionopt ) statement
  for ( init-statementopt for-range-declaration : for-range-initializer ) statement

for-range-declaration:
  attribute-specifier-seqopt decl-specifier-seq declarator
  attribute-specifier-seqopt decl-specifier-seq ref-qualifieropt [ identifier-list ]

for-range-initializer:
  expr-or-braced-init-list

jump-statement:
  break ;
  continue ;
  return expr-or-braced-init-listopt ;
  coroutine-return-statement
  goto identifier ;

coroutine-return-statement:
  co_return expr-or-braced-init-listopt ;

declaration-statement:
  block-declaration

A.6 Declarations

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  nodeclspec-function-declaration
  function-definition
  template-declaration
  deduction-guide
  explicit-instantiation
  explicit-specialization
  export-declaration
  linkage-specification
  namespace-definition
  empty-declaration
  attribute-declaration
block-declaration:
  simple-declaration
  asm-definition
  namespace-alias-definition
  using-declaration
  using-directive
  static_assert-declaration
  alias-declaration
  opaque-enum-declaration

nodeclspec-function-declaration:
  attribute-specifier-seqopt declarator;

alias-declaration:
  using identifier attribute-specifier-seqopt = defining-type-id;

class-declaration:
  decl-specifier-seq init-declarator-listopt ;
  attribute-specifier-seq decl-specifier-seq init-declarator-listopt ;
  attribute-specifier-seqopt decl-specifier-seq ref-qualifieropt [ identifier-list ] initializer;

static_assert-declaration:
  static_assert ( constant-expression ) ;
  static_assert ( constant-expression , string-literal ) ;

empty-declaration:
  ;

attribute-declaration:
  attribute-specifier-seqopt;

decl-specifier:
  storage-class-specifier
  defining-type-specifier
  function-specifier
  friend
typedef
cnstexpr
cnstexpr
inline
decl-specifier-seq:
  decl-specifier attribute-specifier-seqopt
  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
evaluated-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-seq_opt
  defining-type-specifier defining-type-specifier-seq

simple-type-specifier:
  nested-name-specifier_opt type-name
  nested-name-specifier template simple-template-id
  decltype-specifier
  placeholder-type-specifier
  nested-name-specifier_opt template-name
  char
  char8_t
  char16_t
  char32_t
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void

type-name:
  class-name
  enum-name
  typedef-name

elaborated-type-specifier:
  class-key attribute-specifier-seq_opt nested-name-specifier_opt identifier
  class-key simple-template-id
  class-key nested-name-specifier template_opt simple-template-id
  enum nested-name-specifier_opt template-name

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:
  \rightarrow 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-seq_opt
  ...
parameter-declaration-clause:
  parameter-declaration-list_opt ........ opt
  parameter-declaration-list , ........
parameter-declaration-list:
  parameter-declaration
  parameter-declaration-list , parameter-declaration
parameter-declaration:
  attribute-specifier-seq_opt decl-specifier-seq_decl
  attribute-specifier-seq_opt decl-specifier-seq_decl = initializer-clause
  attribute-specifier-seq_opt decl-specifier-seq abstract-declarator_opt
  attribute-specifier-seq_opt decl-specifier-seq abstract-declarator_opt = 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-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 ;
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
tenum-base:
  : type-specifier-seq
tenumerator-list:
  enumerator-definition
tenumerator-list , enumerator-definition
tenumerator-definition:
  enumerator
  enumerator = constant-expression
tenumerator:
  identifier attribute-specifier-seq_opt
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:
  typename\_opt nested-name-specifier unqualified-id

asm-definition:
  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 ] ]
  contract-attribute-specifier
  alignment-specifier

alignment-specifier:
  alignas ( type-id \, .
  alignas ( constant-expression \, .

attribute-using-prefix:
  using attribute-name-space :

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-seqopt )

balanced-token-seq:
  balanced-token
  balanced-token-seq balanced-token

balanced-token:
  ( balanced-token-seqopt )
  [ balanced-token-seqopt ]
  { balanced-token-seqopt }
  any token other than a parenthesis, a bracket, or a brace

contract-attribute-specifier:
  [ [ expects contract-levelopt : conditional-expression ] ]
  [ [ ensures contract-levelopt identifieropt : conditional-expression ] ]
  [ [ assert contract-levelopt : conditional-expression ] ]

contract-level:
  default
  audit
  axiom

A.7 Modules

module-declaration:
  exportopt module module-name module-partitionopt attribute-specifier-seqopt ;

module-name:
  module-name-qualifieropt identifier

module-partition:
  : module-name-qualifieropt identifier

module-name-qualifier:
  identifier .
  module-name-qualifier identifier .

export-declaration:
  export declaration
  export { declaration-seqopt }

module-import-declaration:
  exportopt import module-name attribute-specifier-seqopt ;
  exportopt import module-partition attribute-specifier-seqopt ;
  exportopt import header-name attribute-specifier-seqopt ;

global-module-fragment:
  module ; top-level-declaration-seqopt

A.8 Classes

class-name:
  identifier
  simple-template-id

class-specifier:
  class-head { member-specificationopt }

class-head:
  class-key attribute-specifier-seqopt class-head-name class-virt-specifieropt base-clauseopt
  class-key attribute-specifier-seqopt base-clauseopt

class-head-name:
  nested-name-specifieropt 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
  static_assert-declaration
  template-declaration
  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
decltype-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-listopt )
  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-clauseopt

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

concept-definition:
  concept concept-name = constraint-expression ;

concept-name:
  identifier
type-constraint:
  nested-name-specifieropt concept-name
  nested-name-specifieropt concept-name < template-argument-listopt >
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

context-key:
  class
typename

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

typename-specifier:
  typename nested-name-specifier identifier
  typename nested-name-specifier template<opt > simple-template-id

explicit-instantiation:
  extern<opt > template declaration

explicit-specialization:
  template < > declaration
deduction-guide:
  explicit<opt > template-name ( parameter-declaration-clause ) -> simple-template-id ;

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

group:
  group-part
group group-part

group-part:
  control-line
  if-section
  text-line
  # conditionally-supported-directive

control-line:
  # include pp-tokens new-line
  # define identifier replacement-list new-line
  # define identifier (paren identifier-listopt ) 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-tokensopt new-line
  # new-line

if-section:
  if-group elif-groups opt else-group opt endif-line

if-group:
  # if constant-expression new-line groupopt
  # ifdef identifier new-line groupopt
  # ifndef identifier new-line groupopt

elif-groups:
  elif-group
ehelif-groups elif-group

elif-group:
  # elif constant-expression new-line groupopt

else-group:
  # else new-line groupopt

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-tokensopt

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-global-module-fragment:
   module ; pp-balanced-token-seq module

import-seq:
   top-level-token-seq opt export opt import

top-level-token-seq:
   any pp-balanced-token-seq ending in ; or }

pp-import:
   import header-name pp-import-suffixopt ;
   import header-name-tokens pp-import-suffixopt ;

pp-import-suffix:
   pp-import-suffix-token
   pp-import-suffix pp-import-suffix-token

pp-import-suffix-token:
   any pp-balanced-token other than ;

pp-balanced-token-seq:
   pp-balanced-token
   pp-balanced-token-seq pp-balanced-token

pp-balanced-token:
   pp-ldelim pp-balanced-token-seqopt pp-rdelim
   any preprocessing-token other than a pp-ldelim or pp-rdelim

pp-ldelim: one of
   ( [ { <: <%

pp-rdelim: one of
   ] } ] ) :> %>

va-opt-replacement:
   __VA_OPT__ ( pp-tokensopt )
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.

(2.1) Nesting levels of compound statements (8.3), iteration control structures (8.5), and selection control structures (8.4) [256].

(2.2) Nesting levels of conditional inclusion (15.1) [256].

(2.3) Pointer (9.2.3.1), array (9.2.3.4), and function (9.2.3.5) declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].

(2.4) Nesting levels of parenthesized expressions (7.5.3) within a full-expression [256].

(2.5) Number of characters in an internal identifier (5.10) or macro name (15.5) [1024].

(2.6) Number of characters in an external identifier (5.10, 6.5) [1024].

(2.7) External identifiers (6.5) in one translation unit [65536].

(2.8) Identifiers with block scope declared in one block (6.3.3) [1024].

(2.9) Structured bindings (9.5) introduced in one declaration [256].

(2.10) Macro identifiers (15.5) simultaneously defined in one translation unit [65536].

(2.11) Parameters in one function definition (9.4.1) [256].

(2.12) Arguments in one function call (7.6.1.2) [256].

(2.13) Parameters in one macro definition (15.5) [256].

(2.14) Arguments in one macro invocation (15.5) [256].

(2.15) Characters in one logical source line (5.2) [65536].

(2.16) Characters in a string literal (5.13.5) (after concatenation (5.2)) [65536].

(2.17) Size of an object (6.6.2) [262144].

(2.18) Nesting levels for #include files (15.2) [256].

(2.19) Case labels for a switch statement (8.4.2) (excluding those for any nested switch statements) [16384].

(2.20) Non-static data members (including inherited ones) in a single class (11.3) [16384].

(2.21) Lambda-captures in one lambda-expression (7.5.5.2) [256].

(2.22) Enumeration constants in a single enumeration (9.6) [4096].

(2.23) Levels of nested class definitions (11.3.10) in a single member-specification [256].

(2.24) Functions registered by atexit() (17.5) [32].

(2.25) Functions registered by at_quick_exit() (17.5) [32].

(2.26) Direct and indirect base classes (11.6) [16384].

(2.27) Direct base classes for a single class (11.6) [1024].

(2.28) Class members declared in a single member-specification (including member functions) (11.3) [4096].

(2.29) Final overriding virtual functions in a class, accessible or not (11.6.2) [16384].

(2.30) Direct and indirect virtual bases of a class (11.6.1) [1024].
(2.31) — Static data members of a class (11.3.8.2) [1024].
(2.32) — Friend declarations in a class (11.8.3) [4096].
(2.33) — Access control declarations in a class (11.8.1) [4096].
(2.34) — Member initializers in a constructor definition (11.9.2) [6144].
(2.35) — \texttt{initializer-clauses} in one \texttt{braced-init-list} (9.3) [16384].
(2.36) — Scope qualifications of one identifier (7.5.4.2) [256].
(2.37) — Nested \texttt{linkage-specifications} (9.10) [1024].
(2.38) — Recursive constexpr function invocations (9.1.5) [512].
(2.39) — Full-expressions evaluated within a core constant expression (7.7) [1048576].
(2.40) — Template parameters in a template declaration (13.1) [1024].
(2.41) — Recursively nested template instantiations (13.8.1), including substitution during template argument
deduction (13.9.2) [1024].
(2.42) — Handlers per try block (14.3) [256].
(2.43) — Number of placeholders (20.14.14.4) [10].
Annex C  (informative)
Compatibility  [diff]

C.1  C++ and ISO C  [diff.iso]

This subclause lists the differences between C++ and ISO C, by the chapters of this document.

C.1.1  Clause 5: lexical conventions  [diff.lex]

1  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.

2  Affected subclause: 5.13.3
Change: Type of character literal is changed from int to char.
Rationale: This is needed for improved overloaded function argument type matching. For example:

```c
int function( int i );
int function( char c );

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.

3  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";
void f(char*) {
    char* p = (char*)"abc";
    f(p);
    f((char*)"def");
}
```

How widely used: Programs that have a legitimate reason to treat string literals as pointers to potentially
modifiable memory are probably rare.
C.1.2 Clause 6: basics

Affected subclause: 6.1
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 static objects, 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 static objects must invoke a function call to achieve the initialization.
How widely used: Seldom.

2
Affected subclause: 6.3
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: 6.5 [also 9.1.7]
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: 6.8.3.1
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: 6.7
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.1.3 Clause 7: expressions

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:

    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>) is ill-formed in this International Standard.

4 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*)(struct x {int i;} *)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.

5 Affected subclauses: 7.6.16, 7.6.19, and 7.6.20
Change: The result of a conditional expression, an assignment expression, or a comma expression may be an lvalue.

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,

    char arr[100];
    sizeof(0, arr)

yields 100 in C++ and sizeof(char*) in C.

Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.

How widely used: Rare.

C.1.4 Clause 8: statements

1 Affected subclauses: 8.4.2 and 8.6.4
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.

2 **Affected subclause:** 8.6.3

**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.1.5 Clause 9: declarations

1 **Affected subclause:** 9.1.1

**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:

```c
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:** 9.1.1

**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:** 9.1.3

**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:

```c
typedef struct name1 { /* ... */ } name1; // valid C and C++
struct name { /* ... */ }; // valid C, invalid C++
typedef int name; // valid C, invalid C++
```

**Rationale:** For ease of use, C++ doesn’t require that a type name be prefixed with the keywords class, struct or union when used in object declarations or type casts.

Example:

```c
class name { /* ... */ }; // i has type class name
```
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: 9.1.7 [see also 6.5]
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: 9.1.7
Change: Banning implicit int.
In C++ a decl-specifier-seq must contain a type-specifier, unless it is followed by a declarator for a constructor, a destructor, or a conversion function. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```c
void f(const parm); void f(const int parm);
const n = 3; const int n = 3;
main() int main()
/* ... */ /* ... */
```

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. Could be automated.
How widely used: Common.

6 Affected subclause: 9.1.7.5
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: 9.2.3.5
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.
Example:
```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 “obsolescent” in C.
Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
How widely used: Common.
Affected subclause: 9.2.3.5 [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.

Example:

```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.

Affected subclause: 9.4
Change: In C++, the syntax for function definition excludes the “old-style” C function. In C, “old-style” syntax is allowed, but deprecated as “obsolescent”.

Rationale: Prototypes are essential to type safety.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Syntactic transformation.

How widely used: Common in old programs, but already known to be obsolescent.

Affected subclause: 9.3.1
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++
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: 9.3.2
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";
```

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.
12 **Affected subclause:** 9.6  
**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.

13 **Affected subclause:** 9.6  
**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.1.6 Clause 11: classes

1 **Affected subclause:** 11.2 [see also 9.1.3]  
**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.

2 **Affected subclause:** 11.3.4.2  
**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:
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: 11.3.9

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: 11.3.10

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 6.3.

How widely used: Seldom.

Affected subclause: 11.3.11

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.1.7 Clause 15: preprocessing directives  [diff.cpp]

1 Affected subclause: 15.10

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.2 C++ and ISO C++ 2003  [diff.cpp03]

1 This subclause lists the differences between C++ and ISO C++ 2003 (ISO/IEC 14882:2003, Programming Languages — C++), by the chapters of this document.

C.2.1 Clause 5: lexical conventions  [diff.cpp03.lex]

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:

#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:

#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.

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.2.2 Clause 7: expressions

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:
   
   ```cpp
   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 or produce different results in this International Standard. For example:
   
   ```cpp
   bool b1 = new int && false;          // previously false, now ill-formed
   struct S { operator int(); };       // previously true, now ill-formed
   bool b2 = &S::operator int && false; // previously false, now ill-formed
   ```

C.2.3 Clause 9: declarations

1 Affected subclause: 9.1
   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: 9.3.4
   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:
   ```cpp
   int x[] = { 2.0 };
   ```

C.2.4 Clause 11: classes

1 Affected subclauses: 11.3.4.1, 11.3.6, 11.3.4.2, and 11.3.5
   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: 11.3.6
   Change: User-declared destructors have an implicit exception specification.
   Rationale: Clarification of destructor requirements.
   Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. In particular, destructors that throw exceptions will call std::terminate (without calling std::unexpected) if their exception specification is non-throwing.
C.2.5 Clause 13: templates

1. Affected subclause: 13.1
   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: 13.3
   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 { };  // C++ 2003
template <int N> struct Y { };  // C++ 2003
X< Y< 1 >> 2 > > x;  // C++ 2003
```

3. Affected subclause: 13.7.4.2
   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.2.6 Clause 16: library introduction

1. Affected: Clause 16 – Clause 32
   Change: New reserved identifiers.
   Rationale: Required by new features.
   Effect on original feature: Valid C++ 2003 code that uses any identifiers added to the C++ standard library by 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>, <atomic>, <chrono>, <codecvt>, <condition_variable>, <forward_list>, <future>, <initializer_list>, <mutex>, <random>, <ratio>, <regex>, <scoped_allocator>, <system_error>, <thread>, <tuple>, <typeindex>, <type_traits>, <unordered_map>, and <unordered_set>. In addition the following C compatibility headers are new: <cfenv>, <cinttypes>, <cstdint>, and <cuchar>. 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> for swap.
   Effect on original feature: Valid C++ 2003 code that has been compiled expecting swap to be in <algorithm> may have to instead include <utility>.

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.

5. 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.2.7 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.2.8 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.2.9 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.

C.2.10 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.

C.2.11 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.

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;`
not all containers are empty after construction (array);
— not all containers have constant complexity for swap() (array).

Affected subclause: 22.2
Change: Requirements change: default constructible.
Rationale: Clarification of container requirements.
Effect on original feature: Valid C++ 2003 code that attempts to explicitly instantiate a container using a user-defined type with no default constructor may fail to compile.

Affected subclauses: 22.2.3 and 22.2.6
Change: Signature changes: from void return types.
Rationale: Old signature threw away useful information that may be expensive to recalculate.
Effect on original feature: The following member functions have changed:
— erase(iter) for set, multiset, map, multimap
— erase(begin, end) for set, multiset, map, multimap
— insert(pos, num, val) for vector, deque, list, forward_list
— 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.

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:
— insert(iter, val) for vector, deque, list, set, multiset, map, multimap
— insert(pos, beg, end) for vector, deque, list, forward_list
— erase(begin, end) for set, multiset, map, multimap
— all forms of list::splice
— all forms of list::merge

Valid C++ 2003 code that uses these functions may fail to compile with this International Standard.

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.2.12 Clause 25: algorithms library

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.2.13 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.2.14 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:

```cpp
#include <iostream>

int main() {
    int flag = std::ios_base::hex;
    std::cout.setf(flag);    // error: setf does not take argument of type int
}
```

C.3 C++ and ISO C++ 2011

This subclause lists the differences between C++ and ISO C++ 2011 (ISO/IEC 14882:2011, Programming Languages — C++), 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:

```cpp
#define M(x, ...) __VA_ARGS__
int x[2] = { M(1'2,3'4, 5) };  // error: setf does not take argument of type int
```

C.3.2 Clause 6: basics

Affected subclause: 6.6.5.4.2

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:

```cpp
void* operator new(std::size_t, std::size_t);
void operator delete(void*, std::size_t) noexcept;
```
N4810

© ISO/IEC

In this International Standard, however, the declaration of operator delete might match a predefined usual
(non-placement) operator delete (6.6.5.4). If so, the program is ill-formed, as it was for class member
allocation functions and deallocation functions (7.6.2.5).

C.3.3
1

Clause 7: expressions

[diff.cpp11.expr]

Affected subclause: 7.6.16
Change: A conditional expression with a throw expression as its second or third operand keeps the type
and value category of the other operand.
Rationale: Formerly mandated conversions (lvalue-to-rvalue (7.3.1), array-to-pointer (7.3.2), and functionto-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:
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.
sizeof(true ? "" : throw 0)

In C++ 2011, the expression yields sizeof(const char*).
sizeof(const char[1]).

C.3.4
1

In this International Standard, it yields

Clause 9: declarations

[diff.cpp11.dcl.dcl]

Affected subclause: 9.1.5
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:
struct S {
constexpr const int &f();
int &f();
};

2

Affected subclause: 9.3.1
Change: Classes with default member initializers can be aggregates.
Rationale: Necessary to allow default member initializers to be used by aggregate initialization.
Effect on original feature: Valid C++ 2011 code may fail to compile or may change meaning in this
International Standard. For example:
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
1

Clause 16: library introduction

[diff.cpp11.library]

Affected subclause: 16.5.1.2
Change: New header.
Rationale: New functionality.
§ C.3.5

1582


Effect on original feature: The C++ header `<shared_mutex>` 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++ 2014

1 This subclause lists the differences between C++ and ISO C++ 2014 (ISO/IEC 14882:2014, Programming Languages — C++), by the chapters of this document.

C.4.1 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 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 `0e1_p+0` are three separate tokens each in C++ 2014, but one single token in this International Standard. For example:

```cpp
#define F(a) b ## a
int b0p = F(0p+0);
// ill-formed; equivalent to “int b0p = b0p + 0;” in C++ 2014
```

C.4.2 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: 7.6.2.5 and 7.6.2.6
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.4.3 Clause 9: declarations

1 Affected subclause: 9.1.1
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: 9.1.7.5
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:
3 Affected subclause: 9.2.3.5
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: 9.3.1
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.4.4 Clause 11: classes

1 Affected subclause: 11.9.3
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.4.5 Clause 13: templates

1 Affected subclause: 13.9.2.5
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 *);
```
void bar(A<0> *p) {
    foo(p); // ill-formed; previously well-formed
}

C.4.6 Clause 14: exception handling

Affected subclause: 14.4
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 \texttt{terminate} rather than \texttt{unexpected} and might not perform stack unwinding prior to such a call.

C.4.7 Clause 16: library introduction

Affected subclause: 16.5.1.2
Change: New headers.
Rationale: New functionality.
Effect on original feature: The following \texttt{c++} headers are new: \texttt{<any>}, \texttt{<charconv>}, \texttt{<execution>}, \texttt{<filesystem>}, \texttt{<memory_resource>}, \texttt{<optional>}, \texttt{<string_view>}, and \texttt{<variant>}. Valid C++ 2014 code that \texttt{#include}s 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 \texttt{std} followed by an arbitrary sequence of digits is reserved for future standardization. Valid C++ 2014 code that uses such a top-level namespace, e.g., \texttt{std2}, may be invalid in this International Standard.

C.4.8 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 \texttt{std::function} with an allocator is ill-formed and uses-allocator construction will not pass an allocator to \texttt{std::function} constructors in this International Standard.

Affected subclause: 20.11.3
Change: Different constraint on conversions from \texttt{unique_ptr}.
Rationale: Adding array support to \texttt{shared_ptr}, via the syntax \texttt{shared_ptr<T[]>} and \texttt{shared_ptr<T[N]>}.\nEffect on original feature: Valid C++ 2014 code may fail to compile or may change meaning in this International Standard. For example:

\[
\begin{align*}
#include & \; <memory> \\
std::unique_ptr<int[]> & arr(new int[1]); \\
std::shared_ptr<int> & ptr(std::move(arr)); \; // error: int(*)[] is not compatible with int*
\end{align*}
\]

C.4.9 Clause 21: strings library

Affected subclause: 21.3.2
Change: Non-const \texttt{.data()} member added.
Rationale: The lack of a non-const \texttt{.data()} differed from the similar member of \texttt{std::vector}. This change regularizes behavior for this International Standard.
Effect on original feature: Overloaded functions which have differing code paths for \texttt{char*} and \texttt{const char*} arguments will execute differently when called with a non-const string’s \texttt{.data()} member in this International Standard.

\[
\begin{align*}
\texttt{int f(char *)} & = \texttt{delete;}
\end{align*}
\]
int f(const char *);  
string s;  
int x = f(s.data()); // ill-formed; previously well-formed

C.4.10 Clause 22: containers library  

1 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.4.11 Annex D: compatibility features  

1 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.5 C++ and ISO C++ 2017  

1 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.5.1 Clause 5: lexical conventions  

1 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:

```cpp
#include<types> class import {};  
import<int> f(); // ill-formed; previously well-formed  
::import<int> g(); // OK  
```

2 Affected subclause: 5.11  
Change: New keywords. 
Rationale: Required for new features.

- The char8_t keyword is added to differentiate the types of ordinary and UTF-8 literals (5.13.5). 
- The concept keyword is added to enable the definition of concepts (13.6.8). 
- The constexpr keyword is added to declare immediate functions (9.1.5).
The `co_await`, `co_yield`, and `co_return` keywords are added to enable the definition of coroutines (9.4.4).

The `requires` keyword is added to introduce constraints through a `requires-clause` or a `requires-expression`.

**Effect on original feature:** Valid ISO C++ 2017 code using `char8_t`, `concept`, `conseval`, `co_await`, `co_yield`, `co_return`, or `requires` as an identifier is not valid in this International Standard.

3 **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:

   ```cpp
   namespace N {
   struct X {}
   bool operator<=(X, X);
   template<bool(X, X)> struct Y {}
   Y<operator<=> y;         // ill-formed; previously well-formed
   }
   ```

4 **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 ISO 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:

   ```cpp
   const auto *u8s = u8"text";        // u8s previously deduced as const char*; now deduced as const char8_t*
   const char *ps = u8s;              // ill-formed; previously well-formed
   auto u8c = u8'C';                  // u8c previously deduced as char; now deduced as char8_t
   char *pc = &u8c;                   // ill-formed; previously well-formed
   std::string s = u8"text";         // ill-formed; previously well-formed
   void f(const char *s);
   f(u8"text");                      // ill-formed; previously well-formed
   ```

C.5.2 Clause 6: basics

1 **Affected subclauses:** 6.5, 10.1, and 10.3
   **Change:** New identifiers with special meaning.
   **Rationale:** Required for new features.
   **Effect on original feature:** Top-level declarations beginning with `module` or `import` may be either ill-formed or interpreted differently in this International Standard. [Example:

   ```cpp
   class module;
   module *m1;        // ill-formed; previously well-formed
   ::module *m2;      // OK
   ```

   ```cpp
   class import {};
   import j1;        // was variable declaration; now import-declaration
   ::import j2;      // variable declaration
   ```

2 **Affected subclause:** 6.8.2.1
   **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_-relaxed` store to the same variable by the same thread, then reading the latter value with a `memory_order_-acquire` load no longer provides any “happens before” guarantees, even in the absence of intervening stores by another thread.

C.5.3 Clause 7: expressions

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.5.4 Clause 9: declarations

Affected subclause: 9.3.1
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.

```cpp
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
```

C.5.5 Clause 11: classes

Affected subclauses: 11.3.4 and 11.3.7.2
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:

§ C.5.5

1588
struct S {
    explicit (S)(const S&);    // ill-formed; previously well-formed
    explicit (operator int)();  // ill-formed; previously well-formed
    explicit(true) (S)(int);    // OK
};

2 Affected subclauses: 11.3.4 and 11.3.6
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
};
```
fail to compile: `<ccomplex>`, `<ciso646>`, `<cstdalign>`, `<cstdbool>`, and `<ctgmath>`. To retain the same behavior:

1. a `#include` of `<ccomplex>` can be replaced by a `#include` of `<complex>` (26.4.1),
2. a `#include` of `<ctgmath>` can be replaced by a `#include` of `<cmath>` (26.8.1) and a `#include` of `<complex>`, and
3. a `#include` of `<ciso646>`, `<cstdalign>`, or `<cstdbool>` can simply be removed.

C.5.9 Clause 22: containers library

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.5.10 Clause 25: algorithms library

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.5.11 Clause 29: input/output library

Affected subclauses: 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:

```c++
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
```

Effect on original feature: Valid ISO C++ 2017 code that passes UTF-8 literals to `basic_ostream::operator<<` no longer calls character-related overloads.

```c++
std::cout << u8"text"; // previously called operator<<(const char*) and printed a string;
// now calls operator<<(const void*) and prints a pointer value
std::cout << u8'X'; // previously called operator<<(char) and printed a character;
// now calls operator<<(int) and prints an integer value
```

Effect on original feature: Valid ISO 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.

```c++
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.5.12 Annex D: compatibility features

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.

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 calls `std::uncaught_exception` may fail to compile. It might be revised to use `std::uncaught_exceptions` instead, for clear and portable semantics.

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.

Change: Remove redundant members from `std::allocator`.

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.

Change: Remove redundant members from `std::allocator`.

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.

Change: Remove temporary buffers API.

Rationale: The temporary buffer facility was intended to provide an efficient optimization for small memory requests, but there is little evidence this was achieved in practice, while requiring the user to provide their own exception-safe wrappers to guard use of the facility in many cases.

Effect on original feature: A valid C++ 2017 program that calls `get_temporary_buffer` or `return_temporary_buffer` may fail to compile.

Change: Remove `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.

Change: Remove temporary buffers API.

Rationale: The temporary buffer facility was intended to provide an efficient optimization for small memory requests, but there is little evidence this was achieved in practice, while requiring the user to provide their own exception-safe wrappers to guard use of the facility in many cases.

Effect on original feature: A valid C++ 2017 program that calls `get_temporary_buffer` or `return_temporary_buffer` may fail to compile.

Change: Remove `shared_ptr::unique`.

Rationale: The result of a call to this member function is not reliable in the presence of multiple threads and weak pointers. The member function `use_count` is similarly unreliable, but has a clearer contract in such cases, and remains available for well defined use in single-threaded cases.

Effect on original feature: A valid C++ 2017 program that calls `unique` on a `shared_ptr` object may fail to compile.

Affected subclause: D.9

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.6 C standard library

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.6, 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 names defined in <uchar> (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 type name defined in any of <cstdlib> (17.2.2), <cstring> (21.5.3), 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_eq, not, 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.6.2).

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.6.3).

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.6.4).

C.6.2.7 Macro NULL

The macro NULL, defined in any of <locale> (28.5), <cstdlib> (17.2.1), <stdio> (29.12.1), <stdlib> (17.2.2), <cstring> (21.5.3), <ctime> (27.13), 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

Header <cstring> (21.5.3): The following functions have different declarations:

- strchr
- strpbrk
- strrchr
- strstr
- memchr

Subclause 21.5.3 describes the changes.

Header <cwchar> (21.5.4): The following functions have different declarations:

- wcschr
- wcspbrk
- wcsrchr
- wcsstr
- wmemchr

Subclause 21.5.4 describes the changes.

Header <cstdlib> (17.2.1) declares the name nullptr_t in addition to the names declared in <stddef.h> in the C standard library.
C.6.4 Modifications to behavior

Header `<cstdlib>` (17.2.2): The following functions have different behavior:

1. `atexit`
2. `exit`
3. `abort`

Subclause 17.5 describes the changes.

Header `<csetjmp>` (17.13.2): The following functions have different behavior:

1. `longjmp`

Subclause 17.13.2 describes the changes.

C.6.4.1 Macro `offsetof(type, member-designator)`

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

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

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 (9.11.5).

D.1 Arithmetic conversion on enumerations  [depr.arith.conv.enum]
1 The ability to apply the usual arithmetic conversions (7.4) on operands where one is of enumeration type and the other is of a different enumeration type or a floating-point type is deprecated. [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; // ill-formed  
—end example]
```

D.2 Implicit capture of *this by reference  [depr.capture.this]
1 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
  }
};  
—end example]
```

D.3 Array comparisons  [depr.array.comp]
1 Equality and relational comparisons (7.6.10, 7.6.9) between two operands of array type are deprecated. [Note: 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],  
auto cmp = arr1 <=> arr2; // ill-formed  
—end example]
```

D.4 Redeclaration of static constexpr data members  [depr.static.constexpr]
1 For compatibility with prior C++ International Standards, a constexpr static data member may be redundantly reddeclared outside the class with no initializer. This usage is deprecated. [Example:

```cpp
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.5 Implicit declaration of copy functions

The implicit definition of a copy constructor (11.3.4.2) as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor (11.3.6). The implicit definition of a copy assignment operator (11.3.5) 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 (9.4).

D.6 C standard library headers

For compatibility with the C standard library, the C++ standard library provides the C headers shown in Table 135.

<table>
<thead>
<tr>
<th>Table 135 — C headers</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;assert.h&gt;</td>
</tr>
<tr>
<td>&lt;complex.h&gt;</td>
</tr>
<tr>
<td>&lt;ctype.h&gt;</td>
</tr>
<tr>
<td>&lt;errno.h&gt;</td>
</tr>
<tr>
<td>&lt;float.h&gt;</td>
</tr>
</tbody>
</table>

D.6.1 Header <complex.h> synopsis

#include <complex>

The header <complex.h> behaves as if it simply includes the header <complex> (26.4.1).

[Note: Names introduced by <complex> in namespace std are not placed into the global namespace scope by <complex.h>. — end note]

D.6.2 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.6.3 Header <stdalign.h> synopsis

#define __alignas_is_defined 1

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.6.4 Header <stdbool.h> synopsis

#define __bool_true_false_are_defined 1

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.6.5 Header <tgmath.h> synopsis

#include <cmath>
#include <complex>

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]

[Note: Names introduced by <cmath> or <complex> in namespace std are not placed into the global namespace scope by <tgmath.h>. — end note]
D.6.6 Other C headers

Every C header other than `<complex.h>`, `<iso646.h>`, `<stdalign.h>`, `<stdbool.h>`, and `<tgmath.h>`, 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 `<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 (6.3.6) of the namespace `std` and are then injected into the global namespace scope by explicit `using-declarations` (9.8).

Example: The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — end example

D.7 Relational operators

The header `<utility>` 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&);
}
```

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);

Requires: Type T is `Cpp17EqualityComparable` (Table 23).

Returns: !(x == y).

template<class T> bool operator>(const T& x, const T& y);

Requires: Type T is `Cpp17LessThanComparable` (Table 24).

Returns: y < x.

template<class T> bool operator<=(const T& x, const T& y);

Requires: Type T is `Cpp17LessThanComparable` (Table 24).

Returns: !(y < x).

template<class T> bool operator>=(const T& x, const T& y);

Requires: Type T is `Cpp17LessThanComparable` (Table 24).

Returns: !(x < y).
```

D.8 char* streams

D.8.1 Header `<strstream>` synopsis

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.8.2 Class strstreambuf

```cpp
namespace std {
    class strstreambuf : public basic_streambuf<char> {
    public:
        strstreambuf() : streambuf(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
    static strstate strmode; // exposition only
    streamsize alsize; // exposition only
    void* (*palloc)(size_t); // exposition only
    void (*pfree)(void*); // exposition only
};

1 The class strstreambuf associates the input sequence, and possibly the output sequence, with an object of
some character array type, whose elements store arbitrary values. The array object has several attributes.

2 [Note: 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]

3 [Note: For the sake of exposition, the maintained data is presented here as:

(3.1) — strstate strmode, the attributes of the array object associated with the strstreambuf object;

(3.2) — int alsize, the suggested minimum size for a dynamic array object;

(3.3) — void* (*palloc)(size_t), points to the function to call to allocate a dynamic array object;

(3.4) — void (*pfree)(void*), points to the function to call to free a dynamic array object.
Each object of class `strstreambuf` has a seekable area, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

### D.8.2.1 strstreambuf constructors

#### 1

```c
explicit strstreambuf(streamsize alsize_arg);
```

*Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 136.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsize</td>
<td>alsize_arg</td>
</tr>
<tr>
<td>palloc</td>
<td>a null pointer</td>
</tr>
<tr>
<td>pfree</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

#### 2

```c
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

*Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 137.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsize</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>palloc</td>
<td>palloc_arg</td>
</tr>
<tr>
<td>pfree</td>
<td>pfree_arg</td>
</tr>
</tbody>
</table>

#### 3

```c
strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = nullptr);
strstreambuf(signed char* gnext_arg, streamsize n, signed char* pbeg_arg = nullptr);
strstreambuf(unsigned char* gnext_arg, streamsize n, unsigned char* pbeg_arg = nullptr);
```

*Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 138.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>0</td>
</tr>
<tr>
<td>alsize</td>
<td>an unspecified value</td>
</tr>
<tr>
<td>palloc</td>
<td>a null pointer</td>
</tr>
<tr>
<td>pfree</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

#### 4

`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`.

_for additional information_
If `pbeg_arg` is a null pointer, the function executes:
```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```
Otherwise, the function executes:
```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg, pbeg_arg + N);
```

`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.8.2.3 describes how a dynamically allocated array object is freed.)

### D.8.2.2 Member functions

**void freeze(bool freezel = true);**

**Effects:** If `strmode & dynamic` is nonzero, alters the freeze status of the dynamic array object as follows:

- If `freezel` is true, the function sets `frozen` in `strmode`.
- Otherwise, it clears `frozen` in `strmode`.

**char* str();**

**Effects:** Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

**Remarks:** The return value can be a null pointer.

**int pcount() const;**

**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.8.2.3 strstreambuf overridden virtual functions

**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:

- 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`.

- 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.

To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements `n` to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified. If `palloc` is not a null pointer, the function calls `(*palloc)(n)` to allocate the new dynamic array object. Otherwise, it evaluates the expression `new charT[n]`. In either case, if the allocation fails, the function returns EOF. Otherwise, it sets `allocated` in `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 `(*pfree)(p)`. Otherwise, it evaluates the expression `delete[]p`.

---

332] An implementation should consider `alsize` in making this decision.
If `(strmode & dynamic) == 0`, or if `(strmode & frozen) != 0`, the function cannot extend the array (reallocate it with greater length) to make a write position available.

```c
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:

1. If `c != EOF`, if the input sequence has a putback position available, and if `(char)c == gnext[-1]`, assigns `gnext - 1` to `gnext`.
   Returns `c`.
2. If `c != EOF`, if the input sequence has a putback position available, and if `strmode & constant` is zero, assigns `c` to `*--gnext`.
   Returns `c`.
3. If `c == EOF` and if the input sequence has a putback position available, assigns `gnext - 1` to `gnext`.
   Returns a value other than `EOF`.

Returns `EOF` to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```c
int_type underflow() override;
```

Effects: Reads a character from the input sequence, if possible, without moving the stream position past it, as follows:

1. If the input sequence has a read position available, the function signals success by returning `(unsigned char)*gnext`.
2. Otherwise, if the current write next pointer `pnext` is not a null pointer and is greater than the current read end pointer `gend`, makes a read position available by assigning to `gend` a value greater than `gnext` and no greater than `pnext`.
   Returns `(unsigned char)*gnext`.

Returns `EOF` to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

```c
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 139.

Table 139 — `seekoff` positioning

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(which &amp; ios::in) != 0</code></td>
<td>positions the input sequence</td>
</tr>
<tr>
<td><code>(which &amp; ios::out) != 0</code></td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>`(which &amp; (ios::in</td>
<td>ios::out)) == (ios::in</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 140.

If `(newoff + off) < (seeklow - xbeg)` or `(seekhigh - xbeg) < (newoff + off)`, the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to 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))`.

§ D.8.2.3
Table 140 — newoff values

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>way == ios::beg</td>
<td>0</td>
</tr>
<tr>
<td>way == ios::cur</td>
<td>the next pointer minus the beginning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>way == ios::end</td>
<td>seekhigh minus the beginning pointer (seekhigh - xbeg).</td>
</tr>
</tbody>
</table>

pos_type seekpos(pos_type sp, ios_base::openmode which
= ios_base::in | ios_base::out) override;

17 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;

20 Effects: Behavior is implementation-defined, except that setbuf(0, 0) has no effect.

D.8.3 Class istrstream [depr.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
};

1 The class istrstream supports the reading of objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

(1.1) — sb, the strstreambuf object.

D.8.3.1 istrstream constructors [depr.istrstream.cons]

explicit istrstream(const char* s);
explicit istrstream(char* s);

Effects: Constructs an object of class istrstream, initializing the base class with istream(&sb) and initializing sb with strstreambuf(s,0). s shall designate the first element of an NTBS.

istrstream(const char* s, streamsize n);

Effects: Constructs an object of class istrstream, initializing the base class with istream(&sb) and initializing 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.8.3.2 Member functions

strstreambuf* rdbuf() const;

Returns: const_cast<strstreambuf*>(&sb).

char* str();

Returns: rdbuf()->str().

D.8.4 Class ostrstream

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 freezefl = 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) — sb, the strstreambuf object.

D.8.4.1 ostrstream constructors

ostrstream();

Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb) and initializing sb with strstreambuf().

ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);

Effects: Constructs an object of class ostrstream, initializing the base class with ostream(&sb), and initializing sb with one of two constructors:

(2.1) — If (mode & app) == 0, then s shall designate the first element of an array of n elements. The constructor is strstreambuf(s, n, s).

(2.2) — If (mode & app) != 0, then s shall designate the first element of an array of n elements that contains an NTBS whose first element is designated by s. The constructor is strstreambuf(s, n, s + std::strlen(s)).

§ D.8.4.1 1602

333) The function signature strlen(const char*) is declared in <cstring> (21.5.3).
D.8.4.2 Member functions \[\text{depr.ostrstream.members}\]

```
strstreambuf* rdbuf() const;

\textit{Returns:} (strstreambuf*)\&sb.
```

```
void freeze(bool freezefl = true);

\textit{Effects:} Calls rdbuf()\rightarrow freeze(freezefl).
```

```
char* str();

\textit{Returns:} rdbuf()\rightarrow str().
```

```
int pcount() const;

\textit{Returns:} rdbuf()\rightarrow pcount().
```

D.8.5 Class \texttt{strstream} \[\text{depr.strstream}\]

```
namespace std {

class strstream

\text{:\ public basic_iostream<char> \{ }

\text{\textbf{public:}}

\text{\texttt{// types}}

\text{\texttt{using char_type = char;}}

\text{\texttt{using int_type = char_traits<char>::int_type;}}

\text{\texttt{using pos_type = char_traits<char>::pos_type;}}

\text{\texttt{using off_type = char_traits<char>::off_type;}}

\text{\texttt{// constructors/destructor}}

\text{\texttt{strstream();}}

\text{\texttt{strstream(char* s, int n,}}

\text{\texttt{ios_base::openmode mode = ios_base::in|ios_base::out);}}

\text{\texttt{virtual \texttt{-strstream();}}}

\text{\texttt{// members}}

\text{\texttt{strstreambuf* rdbuf() const;}}

\text{\texttt{void freeze(bool freezefl = true);}}

\text{\texttt{int pcount() const;}}

\text{\texttt{char* str();}}

\text{\texttt{private:}}

\text{\texttt{strstreambuf sb; // exposition only}}
}
```

1 The class \texttt{strstream} supports reading and writing from objects of class \texttt{strstreambuf}. It supplies a \texttt{strstreambuf} object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

\begin{itemize}
\item \texttt{sb}, the \texttt{strstreambuf} object.
\end{itemize}

D.8.5.1 \texttt{strstream} constructors \[\text{depr.strstream.cons}\]

```
strstream();

\textit{Effects:} Constructs an object of class \texttt{strstream}, initializing the base class with \texttt{iostream(&sb)}.
```

```
strstream(char* s, int n,

\texttt{ios_base::openmode mode = ios_base::in|ios_base::out);}

\textit{Effects:} Constructs an object of class \texttt{strstream}, initializing the base class with \texttt{iostream(&sb)} and initializing \texttt{sb} with one of the two constructors:

\begin{itemize}
\item If \texttt{(mode \& app)} == 0, then \texttt{s} shall designate the first element of an array of \texttt{n} elements. The constructor is \texttt{strstreambuf(s,n,s)}.
\item If \texttt{(mode \& app)} != 0, then \texttt{s} shall designate the first element of an array of \texttt{n} elements that contains an \texttt{NTBS} whose first element is designated by \texttt{s}. The constructor is \texttt{strstreambuf(s,n,s + std::strlen(s))}.
\end{itemize}
```
D.8.5.2 strstream destructor
virtual ~strstream();

Effects: Destroys an object of class strstream.

D.8.5.3 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.9 Deprecated type traits

The header <type_traits> 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.10 Deprecated iterator primitives

D.10.1 Basic iterator

The header <iterator> 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:

§ D.10.1 1604
class MyIterator : public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
  // code implementing ++, etc.
};

—end example

D.11 Deprecated move_iterator access [depr.move.iter.elem]

1 The following member is declared in addition to those members specified in 23.5.3.5:

```cpp
namespace std {
  template<class Iterator>
  class move_iterator {
  public:
    constexpr pointer operator->() const;
  }
}
```

constexpr pointer operator->() const;

2 Returns: current.

D.12 Deprecated shared_ptr atomic access [depr.util.smartptr.shared.atomic]

1 The header <memory> has the following additions:

```cpp
namespace std {
  template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);

  template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
  template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

  template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
  shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
  bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
  bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  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.

```cpp
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.
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.

template<class T>
  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 31.7.1.

D.13 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.14 Deprecated standard code conversion facets

The header <codecvt> provides code conversion facets for various character encodings.

D.14.1 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.14.1
D.14.2 Requirements

1 For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:
   
   1.1 Elem is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`.
   
   1.2 Maxcode is the largest wide-character code that the facet will read or write without reporting a conversion error.
   
   1.3 If `(Mode & consume_header)`, the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.
   
   1.4 If `(Mode & generate_header)`, the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.
   
   1.5 If `(Mode & little_endian)`, the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

2 For the facet `codecvt_utf8`:
   
   2.1 The facet shall convert between UTF-8 multibyte sequences and UCS-2 or UTF-32 (depending on the size of Elem) within the program.
   
   2.2 Endianness shall not affect how multibyte sequences are read or written.
   
   2.3 The multibyte sequences may be written as either a text or a binary file.

3 For the facet `codecvt_utf16`:
   
   3.1 The facet shall convert between UTF-16 multibyte sequences and UCS-2 or UTF-32 (depending on the size of Elem) within the program.
   
   3.2 Multibyte sequences shall be read or written according to the Mode flag, as set out above.
   
   3.3 The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

4 For the facet `codecvt_utf8_utf16`:
   
   4.1 The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.
   
   4.2 Endianness shall not affect how multibyte sequences are read or written.
   
   4.3 The multibyte sequences may be written as either a text or a binary file.

5 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.15 Deprecated convenience conversion interfaces

1 The header `<locale>` has the following additions:

```cpp
namespace std {
    template<class Codecvt, class Elem = wchar_t,
             class Wide_alloc = allocator<Elem>,
             class Byte_alloc = allocator<char>>
    class wstring_convert;

    template<class Codecvt, class Elem = wchar_t,
             class Tr = char_traits<Elem>>
    class wbuffer_convert;
}
```

D.15.1 Class template wstring_convert

1 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 Wide_alloc = allocator<Elem>,
            class Byte_alloc = allocator<char>>
    class wstring_convert {
    public:
        using byte_string = basic_string<char, char_traits<char>, Byte_alloc>;
        using wide_string = basic_string<Elem, char_traits<Elem>, Wide_alloc>;
        using state_type = typename Codecvt::state_type;
        using int_type = typename wide_string::traits_type::int_type;

        wstring_convert() : wstring_convert(new Codecvt) {}  
        explicit wstring_convert(Codecvt* pcvt);
        wstring_convert(Codecvt* pcvt, state_type state);
        explicit wstring_convert(const byte_string& byte_err,
                                  const wide_string& wide_err = wide_string());
        ~wstring_convert();

        wstring_convert(const wstring_convert&) = delete;
        wstring_convert& operator=(const wstring_convert&) = delete;

        wide_string from_bytes(char byte);
        wide_string from_bytes(const char* ptr);
        wide_string from_bytes(const byte_string& str);
        wide_string from_bytes(const char* first, const char* last);

        byte_string to_bytes(Elem wchar);
        byte_string to_bytes(const Elem* wptr);
        byte_string to_bytes(const wide_string& wstr);
        byte_string to_bytes(const Elem* first, const Elem* last);

        size_t converted() const noexcept;
        state_type state() const;

    private:
        byte_string byte_err_string; // exposition only
        wide_string wide_err_string; // exposition only
        Codecvt* cvtptr;             // exposition only
        state_type cvtstate;        // exposition only
        size_t cvtcount;            // exposition only
    };
}

2 The class template describes an object that controls conversions between wide string objects of class `basic_string<Elem, char_traits<Elem>, Wide_alloc>` and byte string objects of class `basic_string<char, char_traits<char>, Byte_alloc>`. 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>`.

3 An object of this class template stores:

(3.1) — `byte_err_string` — a byte string to display on errors

(3.2) — `wide_err_string` — a wide string to display on errors

(3.3) — `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
— cvtstate — a conversion state object
— cvtcount — a conversion count

using byte_string = basic_string<char, char_traits<char>, Byte_alloc>;

The type shall be a synonym for basic_string<char, char_traits<char>, Byte_alloc>.

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 int_type = typename wide_string::traits_type::int_type;

The type shall be a synonym for wide_string::traits_type::int_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>, Wide_alloc>;

The type shall be a synonym for basic_string<Elem, char_traits<Elem>, Wide_alloc>.

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());

16    Requires: For the first and second constructors, pcvt != nullptr.
17    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();

18    Effects: The destructor shall delete cvtptr.

D.15.2 Class template wbuffer_convert [depr.conversions.buffer]

1 Class template wbuffer_convert looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template wstring_convert, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

namespace std {
    template<class Codecvt, class Elem = wchar_t, class Tr = char_traits<Elem>>
    class wbuffer_convert : public basic_streambuf<Elem, Tr> { public:
        using state_type = typename Codecvt::state_type;
        wbuffer_convert() : wbuffer_convert(nullptr) {} explicit wbuffer_convert(streambuf* bytebuf,
            Codecvt* pcvt = new Codecvt,
            state_type state = state_type());
        ~wbuffer_convert();
        wbuffer_convert(const wbuffer_convert&) = delete;
        wbuffer_convert& operator=(const wbuffer_convert&) = delete;
        streambuf* rdbuf() const;
        streambuf* rdbuf(streambuf* bytebuf);
        state_type state() const;
    private:
        streambuf* bufptr;       // exposition only
        Codecvt* cvtptr;         // exposition only
        state_type cvtstate;     // exposition only
    };
}

2 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 satisfy the requirements of the standard code-conversion facet codecvt<Elem, char, mbstate_t>.

3 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;

4    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.16 Deprecated locale category facets

1 The ctype locale category includes the following facets as if they were specified in table Table 90 of 28.3.1.1.1.
   codecvt<char16_t, char, mbstate_t>
   codecvt<char32_t, char, mbstate_t>

2 The ctype locale category includes the following facets as if they were specified in table Table 91 of 28.3.1.1.1.
   codecvt_byname<char16_t, char, mbstate_t>
   codecvt_byname<char32_t, char, mbstate_t>

3 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.17 Deprecated filesystem path factory functions

template<class Source>
   path u8path(const Source& source);
template<class InputIterator>
   path u8path(InputIterator first, InputIterator last);
   Requires: The source and [first, last) sequences are UTF-8 encoded. The value type of Source and InputIterator is char. 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));

   § D.17 1612
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]
Bibliography

The following documents are cited informatively in this document.

— 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 (25.9.2) 1084
adjacent.difference (25.9.11) 1091
adjustfield.manip (25.5.6) 1330
alg.3way (25.7.11) 1079
alg.adjacent.find (25.5.8) 1032
alg.all.of (25.5.1) 1027
alg.any.of (25.5.2) 1027
alg.binary.search (25.7.3) 1060
alg.c.library (25.10) 1093
alg.clamp (25.7.9) 1078
alg.copy (25.6.1) 1038
alg.count (25.5.9) 1032
alg.equal (25.5.11) 1034
alg.fill (25.6.6) 1046
alg.find (25.5.5) 1029
alg.find.first.of (25.5.7) 1030
alg.first.of (25.5.7) 1031
alg.foreach (25.5.4) 1028
alg.generate (25.6.7) 1046
alg.heap.operations (25.7.7) 1071
alg.is.permutation (25.5.12) 1036
alg.lex.comparison (25.7.10) 1078
alg.merge (25.7.5) 1064
alg.min.max (25.7.8) 1074
alg.modifying.operations (25.6) 1038
alg.move (25.6.2) 1040
alg.none.of (25.5.3) 1027
alg.nonmodifying (25.5) 1027
alg.nth.element (25.7.2) 1059
alg.partitions (25.7.4) 1062
alg.permutation.generators (25.7.12) 1079
alg.random.sample (25.6.12) 1052
alg.random.shuffle (25.6.13) 1053
alg.remove (25.6.8) 1047
alg.replace (25.6.5) 1044
alg.req (23.3.7) 901
alg.req.general (23.3.7.1) 901
alg.req.ind.cmp (23.3.7.5) 902
alg.req.ind.copy (23.3.7.3) 902
alg.req.ind.move (23.3.7.2) 901
alg.req.ind.swap (23.3.7.4) 902
alg.req.mergeable (23.3.7.7) 903
alg.req.permutable (23.3.7.6) 902
alg.req.sortable (23.3.7.8) 903
alg.reverse (25.6.10) 1050
alg.rotate (25.6.11) 1051
alg.search (25.5.13) 1036
alg.set.operations (25.7.6) 1066
alg.shift (25.6.14) 1053
alg.sort (25.7.1) 1055
alg.sorting (25.7) 1054
alg.swap (25.6.3) 1042
alg.transform (25.6.4) 1042
alg.unique (25.6.9) 1049
algorithm.stable (16.5.5.7) 470
algorithm.syn (25.4) 991
algorithms (Clause 25) 986
algorithms.general (25.1) 986
algorithms.parallel (25.3) 988
algorithms.parallel.defns (25.3.1) 988
algorithms.parallel.exceptions (25.3.4) 990
algorithms.parallel.exec (25.3.3) 989
algorithms.parallel.overloads (25.3.5) 990
algorithms.parallel.user (25.3.2) 989
algorithms.requirements (25.2) 986
alloc.errors (17.6.3) 494
allocator.adaptor (20.13) 654
allocator.adaptor.cnstr (20.13.3) 656
allocator.adaptor.members (20.13.4) 657
allocator.adaptor.syn (20.13.1) 654
allocator.adaptor.types (20.13.2) 656
allocator.globals (20.10.10.2) 611
allocator.members (20.10.10.1) 611
allocator.requirements (16.5.3.5) 459
allocator.requirements.completeness (16.5.3.5.1) 464
allocator.tag (20.10.7) 606
allocator.trait (20.10.9) 609
allocator.trait.members (20.10.9.2) 610
allocator.trait.types (20.10.9.1) 609
allocator.uses (20.10.8) 607
allocator.uses.construction (20.10.8.2) 607
allocator.uses.trait (20.10.8.1) 607
alt.headers (16.5.4.4) 467
any (20.8) 586
any.assign (20.8.3.2) 588
any.cast (20.8.2) 586
any.class (20.8.3) 587
any.default (20.8.3.1) 587
any.modifiers (20.8.3.3) 589
any.nonmembers (20.8.4) 590
any.observers (20.8.3.4) 590
any.synop (20.8.1) 586
arithmetic.operations (20.14.6) 663
arithmetic.operations.divides (20.14.6.4) 664
arithmetic.operations.minus (20.14.6.2) 664
arithmetic.operations.modulus (20.14.6.5) 665
arithmetic.operations.multiples (20.14.6.3) 664
arithmetic.operations.negative (20.14.6.6) 665
arithmetic.operations.plus (20.14.6.1) 664
array (22.3.7) 799
array.cons (22.3.7.2) 800
array.members (22.3.7.3) 800
array.overview (22.3.7.1) 799
array.special (22.3.7.4) 800
array.syn (22.3.2) 796
array.tuple (22.3.7.6) 801
array.zero (22.3.7.5) 800
assertions (19.3) 533
assertions.assert (19.3.2) 533
associative (22.4) 823
associative.general (22.4.1) 823
associative.map.syn (22.4.2) 823
associative.reqmts (22.2.6) 773
associative.reqmts.except (22.2.6.1) 782
associative.set.syn (22.4.3) 824
atomics (Clause 31) 1477
atomics.alias (31.3) 1481
atomics.fences (31.10) 1498
atomics.flag (31.9) 1497
atomics.general (31.1) 1477
atomics.lockfree (31.5) 1482
atomics.nonmembers (31.8) 1497
atomics.order (31.4) 1481
atomics.ref.float (31.6.3) 1486
atomics.ref.generic (31.6) 1483
atomics.ref.int (31.6.2) 1485
atomics.ref.memop (31.6.5) 1488
atomics.ref.operations (31.6.1) 1483
atomics.ref.pointer (31.6.4) 1487
atomics.syn (31.2) 1477
atomics.types.float (31.7.3) 1494
atomics.types.generic (31.7) 1488
atomics.types.int (31.7.2) 1492
atomics.types.memop (31.7.5) 1497
atomics.types.operations (31.7.1) 1489
atomics.types.pointer (31.7.4) 1495

back.insert.iter.ops (23.5.2.1.1) 911
back.insert.iterator (23.5.2.1) 911
back inserter (23.5.2.1.2) 911
bad.alloc (17.6.3.1) 494
bad.cast (17.7.3) 497
bad.exception (17.9.3) 500
bad.typeid (17.7.4) 498
basefield.manip (29.5.6.3) 1330
basic (Clause 6) 24
basic.align (6.6.6) 59
basic.compound (6.7.2) 67
basic.def (6.1) 24
basic.def.ordr (6.2) 26
basic.exec (6.8) 69
basic.fundamental (6.7.1) 65
basic.funscone (6.3.5) 32
basic.indet (6.6.4) 55
basic.ios.cons (29.5.5.2) 1326
basic.ios.members (29.5.5.3) 1326
basic.life (6.6.3) 53
basic.link (6.5) 47

basic.lookup (6.5) 35
basic.lookup.argdep (6.4.2) 39
basic.lookup.classref (6.4.5) 46
basic.lookup.elab (6.4.4) 46
basic.lookup.qual (6.4.3) 41
basic.lookup.udir (6.4.6) 47
basic.lookup.unqual (6.4.1) 35
basic.lval (7.2.1) 82
basic.memobj (6.6) 50
basic.namespace (9.7) 207
basic.scope (6.3) 30
basic.scope.block (6.3.3) 32
basic.scope.class (6.3.7) 33
basic.scope.declarative (6.3.1) 30
basic.scope.enum (6.3.8) 34
basic.scope.hiding (6.3.10) 35
basic.scope.name (6.3.6) 32
basic.scope.param (6.3.4) 32
basic.scope.pdecl (6.3.2) 30
basic.scope.temp (6.3.9) 34
basic.start (6.8.3) 77
basic.start.dynamic (6.8.3.3) 78
basic.start.main (6.8.3.1) 77
basic.start.static (6.8.3.2) 77
basic.start.term (6.8.3.4) 79
basic.t (6.5) 56
basic.tc (6.6.5.3) 57
basic.tc (6.6.5.4) 57
basic.tc (6.6.5.4.1) 57
basic.tc (6.6.5.4.2) 58
basic.tc (6.6.5.4.3) 59
basic.tc.inherit (6.6.5.5) 59
basic.tc.static (6.6.5.1) 56
basic.tc.thread (6.6.5.2) 57
basic.string (21.3.2) 723
basic.string.hash (21.3.5) 745
basic.string.literals (21.3.6) 745
basic.type.qualifier (6.7.3) 68
basic.types (6.7) 63
bidirectional.iterators (23.3.5.5) 899
binary_search (25.7.3.4) 1061
bit (26.5) 1103
bit.cast (26.5.3) 1104
bit.general (26.5.1) 1103
bit.pow.two (26.5.4) 1104
bit.syn (25.6.2) 1103
bitset.types (16.4.2.2.4) 450
bitset (20.9) 591
bitset.cons (20.9.2.1) 593
bitset.hash (20.9.3) 596
bitset.members (20.9.2.2) 593
bitset.operands (20.9.4) 596
bitset.syn (20.9.1) 591
bitwise.operations (20.14.10) 670
bitwise.operations.and (20.14.10.1) 670
bitwise.operations.not (20.14.10.4) 671
bitwise.operations.or (20.14.10.2) 670
bitwise.operations.xor (20.14.10.3) 671
byte.strings (16.4.2.2.5.1) 452

Cross references 1616
Cross references 1621

diff.cpp11.dcl.dcl (C.3.4) 1582
diff.cpp11.expr (C.3.3) 1582
diff.cpp11.input.output (C.3.6) 1583
diff.cpp11.lex (C.3.1) 1581
diff.cpp11.library (C.3.5) 1582
diff.cpp11 (C.4) 1583
diff.cpp11.class (C.4.4) 1584
diff.cpp11.containers (C.4.10) 1586
diff.cpp11.dcl.dcl (C.4.3) 1583
diff.cpp11.depr (C.4.11) 1585
diff.cpp11.except (C.4.6) 1585
diff.cpp11.lex (C.4.1) 1583
diff.cpp11.library (C.4.9) 1585
diff.cpp11.string (C.4.8) 1585
diff.cpp11 (C.5) 1586
diff.cpp11.alg.reqs (C.5.10) 1590
diff.cpp11.class (C.5.2) 1587
diff.cpp11.containers (C.5.9) 1590
diff.cpp11.dcl.dcl (C.5.4) 1588
diff.cpp11.depr (C.5.12) 1590
diff.cpp11.except (C.5.7) 1589
diff.cpp11.lex (C.5.3) 1588
diff.cpp11.input.output (C.5.11) 1590
diff.cpp11.library (C.5.1) 1586
diff.cpp11.temp (C.5.6) 1589
diff.dcl (C.1.5) 1571
diff.expr (C.1.3) 1569
diff.header.assert.h (C.6.2.3) 1592
diff.header.iso646.h (C.6.2.4) 1592
diff.header.stdalign.h (C.6.2.5) 1592
diff.header.stdbool.h (C.6.2.6) 1592
diff.iso (C.1) 1568
diff.lex (C.1.1) 1568
diff.library (C.6) 1591
diff.malloc (C.6.4.2) 1593
diff.mods.to.behavior (C.6.4) 1593
diff.mods.to.declarations (C.6.3) 1592
diff.mods.to.declarations (C.6.2) 1592
diff.mods.to.headers (C.6.1) 1591
diff.null (C.6.2.7) 1592
diff.offsetof (C.6.4.1) 1593
diff.stat (C.1.4) 1570
diff.wchar.t (C.6.2.2) 1592
domain.error (19.2.3) 531

enumerated.types (16.4.2.2.3) 450
equal.range (25.7.3.3) 1061
erroso (19.4) 533
error.reporting (29.5.7) 1331
except (Clause 14) 418
except.ctor (14.2) 420
except.handle (14.3) 421
except.nested (17.9.7) 501
except.spec (14.4) 422
except.special (14.5) 425
except.terminate (14.5.1) 425
except.throw (14.1) 419
except.uncaught (14.5.2) 426
exception (17.9.2) 499
exception.syn (17.9.1) 499
exception.terminate (17.9.4) 500
exclusive.scan (25.9.7) 1088
execpol (20.18) 709
execpol.general (20.18.1) 709
execpol.objects (20.18.8) 711
execpol.par (20.18.5) 710
execpol.parunseq (20.18.6) 710
execpol.seq (20.18.4) 710
execpol.type (20.18.3) 710
execpol.unseq (20.18.7) 711
execution.syn (20.18.2) 709
expos.only.func (16.4.2.1) 450
expos.only.types (16.4.2.2.2) 450
expr (Clause 7) 81
expr.add (7.6.6) 127
expr.alignof (7.6.2.7) 125
expr.arith.conv (7.4) 89
expr.ass (7.6.19) 134
expr.await (7.6.2.3) 116
expr.bit.and (7.6.11) 131
expr.call (7.6.1.2) 106
expr.cast (7.6.3) 125
expr.comma (7.6.20) 135
expr.composite (7.6) 105
expr.cond (7.6.16) 132
expr.const (7.7) 135
expr.const.cast (7.6.1.10) 114
expr.context (7.2.3) 84
expr.delete (7.6.2.6) 123
expr.dynamic.cast (7.6.1.6) 109
expr.eq (7.6.10) 130
expr.log.and (7.6.14) 131
expr.log.or (7.6.15) 131
expr.mptr.oper (7.6.4) 126
expr.mul (7.6.5) 127
expr.new (7.6.2.5) 119
expr.or (7.6.13) 131
expr.post (7.6.1) 105
expr.post.increment (7.6.1.5) 109
expr.pre (7.1) 81
expr.pre.increment (7.6.2.2) 116
expr.prim (7.5) 90
expr.prim.fold (7.5.6) 102
expr.prim.id (7.5.4) 91
expr.prim.idctor (7.5.4.3) 93
expr.prim.idqual (7.5.4.2) 92
expr.prim.id.unqual (7.5.4.1) 92
expr.prim.lambda (7.5.5) 93
expr.prim.lambda.capture (7.5.5.2) 97
expr.prim.lambdaClosure (7.5.5.1) 94
expr.prim.literal (7.5.1) 90
expr.prim.paren (7.5.3) 91
expr.prim.req (7.5.7) 102

Cross references 1621
<table>
<thead>
<tr>
<th>Expression Requirements</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr.prim.req.compound</td>
<td>104</td>
</tr>
<tr>
<td>expr.prim.req.nested</td>
<td>105</td>
</tr>
<tr>
<td>expr.prim.req.simple</td>
<td>104</td>
</tr>
<tr>
<td>expr.prim.req.type</td>
<td>104</td>
</tr>
<tr>
<td>expr.prim.this</td>
<td>90</td>
</tr>
<tr>
<td>expr.prop</td>
<td>82</td>
</tr>
<tr>
<td>expr.ref</td>
<td>108</td>
</tr>
<tr>
<td>expr.reinterpret.cast</td>
<td>113</td>
</tr>
<tr>
<td>expr.shift</td>
<td>128</td>
</tr>
<tr>
<td>expr.sizeof</td>
<td>118</td>
</tr>
<tr>
<td>expr.spaceship</td>
<td>128</td>
</tr>
<tr>
<td>expr.static.cast</td>
<td>111</td>
</tr>
<tr>
<td>expr.sub</td>
<td>106</td>
</tr>
<tr>
<td>expr.throw</td>
<td>134</td>
</tr>
<tr>
<td>expr.type</td>
<td>83</td>
</tr>
<tr>
<td>expr.type.conv</td>
<td>108</td>
</tr>
<tr>
<td>expr.typeid</td>
<td>111</td>
</tr>
<tr>
<td>expr.unary</td>
<td>115</td>
</tr>
<tr>
<td>expr.unary.op</td>
<td>115</td>
</tr>
<tr>
<td>expr.xor</td>
<td>131</td>
</tr>
<tr>
<td>ext.manip</td>
<td>1360</td>
</tr>
<tr>
<td>extern.names</td>
<td>467</td>
</tr>
<tr>
<td>extern.types</td>
<td>467</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Facet</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>facet.ctype.char.dtor</td>
<td>1283</td>
</tr>
<tr>
<td>facet.ctype.char.members</td>
<td>1283</td>
</tr>
<tr>
<td>facet.ctype.charStatics</td>
<td>1284</td>
</tr>
<tr>
<td>facet.ctype.char.virtuels</td>
<td>1284</td>
</tr>
<tr>
<td>facet.ctype.special</td>
<td>1282</td>
</tr>
<tr>
<td>facet.num.get.members</td>
<td>1289</td>
</tr>
<tr>
<td>facet.num.get.virtuels</td>
<td>1289</td>
</tr>
<tr>
<td>facet.num.put.members</td>
<td>1292</td>
</tr>
<tr>
<td>facet.num.put.virtuels</td>
<td>1292</td>
</tr>
<tr>
<td>facet.mumpunct</td>
<td>1295</td>
</tr>
<tr>
<td>facet.mumpunct.members</td>
<td>1295</td>
</tr>
<tr>
<td>facet.mumpunct.virtuels</td>
<td>1296</td>
</tr>
<tr>
<td>file.streams</td>
<td>1371</td>
</tr>
<tr>
<td>filebuf</td>
<td>1372</td>
</tr>
<tr>
<td>filebuf.assign</td>
<td>1374</td>
</tr>
<tr>
<td>filebuf.cons</td>
<td>1373</td>
</tr>
<tr>
<td>filebuf.members</td>
<td>1374</td>
</tr>
<tr>
<td>filebuf.virtuels</td>
<td>1375</td>
</tr>
<tr>
<td>filesystems</td>
<td>1389</td>
</tr>
<tr>
<td>floatfield.manip</td>
<td>1330</td>
</tr>
<tr>
<td>fntflags.manip</td>
<td>1329</td>
</tr>
<tr>
<td>fntflags.state</td>
<td>1321</td>
</tr>
<tr>
<td>forward</td>
<td>547</td>
</tr>
<tr>
<td>forward.iterators</td>
<td>898</td>
</tr>
<tr>
<td>forward.list.erase</td>
<td>810</td>
</tr>
<tr>
<td>forward.list.syn</td>
<td>797</td>
</tr>
<tr>
<td>forwardlist</td>
<td>805</td>
</tr>
<tr>
<td>forwardlist.access</td>
<td>807</td>
</tr>
<tr>
<td>forwardlist.cons</td>
<td>807</td>
</tr>
<tr>
<td>forwardlist.iter</td>
<td>807</td>
</tr>
<tr>
<td>forwardlist.modifiers</td>
<td>807</td>
</tr>
<tr>
<td>forwardlist.ops</td>
<td>809</td>
</tr>
<tr>
<td>forwardlist.overview</td>
<td>805</td>
</tr>
<tr>
<td>fp.style</td>
<td>480</td>
</tr>
<tr>
<td>fpos</td>
<td>1323</td>
</tr>
<tr>
<td>fpos.members</td>
<td>1324</td>
</tr>
<tr>
<td>fpos.operations</td>
<td>1324</td>
</tr>
<tr>
<td>front.insert.iter.ops</td>
<td>912</td>
</tr>
<tr>
<td>front.insert.iterator</td>
<td>911</td>
</tr>
<tr>
<td>front inserter</td>
<td>912</td>
</tr>
<tr>
<td>fs.class.directory.entry</td>
<td>1413</td>
</tr>
<tr>
<td>fs.class.directory.iterator</td>
<td>1417</td>
</tr>
<tr>
<td>fs.class.file.status</td>
<td>1411</td>
</tr>
<tr>
<td>fs.class.filesystem.error</td>
<td>1409</td>
</tr>
<tr>
<td>fs.class.path</td>
<td>1394</td>
</tr>
<tr>
<td>fs.class.rec.dir.itr</td>
<td>1419</td>
</tr>
<tr>
<td>fs.conform.9945</td>
<td>1389</td>
</tr>
<tr>
<td>fs.conform.os</td>
<td>1390</td>
</tr>
<tr>
<td>fs.conformance</td>
<td>1389</td>
</tr>
<tr>
<td>fs.dir.entry.cons</td>
<td>1415</td>
</tr>
<tr>
<td>fs.dir.entry.mods</td>
<td>1415</td>
</tr>
<tr>
<td>fs.dir.entry.obs</td>
<td>1415</td>
</tr>
<tr>
<td>fs.dir.itr.members</td>
<td>1418</td>
</tr>
<tr>
<td>fs.dir.itr.nonmembers</td>
<td>1419</td>
</tr>
<tr>
<td>fs.enum</td>
<td>1410</td>
</tr>
<tr>
<td>fs.enum.copy.opts</td>
<td>1411</td>
</tr>
<tr>
<td>fs.enum.dir.opts</td>
<td>1411</td>
</tr>
<tr>
<td>fs.enum.file.type</td>
<td>1410</td>
</tr>
<tr>
<td>fs.enum.path.format</td>
<td>1410</td>
</tr>
<tr>
<td>fs.enum.perm.opts</td>
<td>1411</td>
</tr>
<tr>
<td>fs.err.report</td>
<td>1393</td>
</tr>
<tr>
<td>fs.file.status.cons</td>
<td>1413</td>
</tr>
<tr>
<td>fs.file.status.mods</td>
<td>1413</td>
</tr>
<tr>
<td>fs.file.status.obs</td>
<td>1413</td>
</tr>
<tr>
<td>fs.filesystem.error.members</td>
<td>1409</td>
</tr>
<tr>
<td>fs.filesystem.syn</td>
<td>1390</td>
</tr>
<tr>
<td>fs.general</td>
<td>1389</td>
</tr>
<tr>
<td>fs.norm.ref</td>
<td>1390</td>
</tr>
<tr>
<td>fs.op.absolute</td>
<td>1422</td>
</tr>
<tr>
<td>fs.op.canonical</td>
<td>1422</td>
</tr>
<tr>
<td>fs.op.copy</td>
<td>1423</td>
</tr>
<tr>
<td>fs.op.copy.file</td>
<td>1424</td>
</tr>
<tr>
<td>fs.op.copy.symlink</td>
<td>1425</td>
</tr>
<tr>
<td>fs.op.create.dir.symlk</td>
<td>1426</td>
</tr>
<tr>
<td>fs.op.create.directories</td>
<td>1425</td>
</tr>
<tr>
<td>fs.op.create.directory</td>
<td>1425</td>
</tr>
<tr>
<td>fs.op.create.hard.lk</td>
<td>1426</td>
</tr>
<tr>
<td>fs.op.create.symlink</td>
<td>1426</td>
</tr>
<tr>
<td>fs.op.current.path</td>
<td>1427</td>
</tr>
<tr>
<td>fs.op.equivalent</td>
<td>1427</td>
</tr>
<tr>
<td>fs.op.exists</td>
<td>1427</td>
</tr>
<tr>
<td>fs.op.file.size</td>
<td>1427</td>
</tr>
<tr>
<td>fs.op.funcs</td>
<td>1422</td>
</tr>
<tr>
<td>fs.op.hard.lk.ct</td>
<td>1428</td>
</tr>
<tr>
<td>fs.op.is.block.file</td>
<td>1428</td>
</tr>
<tr>
<td>fs.op.is.char.file</td>
<td>1428</td>
</tr>
<tr>
<td>fs.op.is.directory</td>
<td>1428</td>
</tr>
<tr>
<td>fs.op.is.empty</td>
<td>1428</td>
</tr>
<tr>
<td>fs.op.is_fifo</td>
<td>1429</td>
</tr>
<tr>
<td>fs.op.is.other</td>
<td>1429</td>
</tr>
<tr>
<td>Function/Method</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td>map.access</td>
<td>829</td>
</tr>
<tr>
<td>map.cons</td>
<td>829</td>
</tr>
<tr>
<td>map.erasure</td>
<td>831</td>
</tr>
<tr>
<td>map.modifiers</td>
<td>829</td>
</tr>
<tr>
<td>map.overview</td>
<td>826</td>
</tr>
<tr>
<td>mask.array.assign</td>
<td>1159</td>
</tr>
<tr>
<td>mask.array.comp.assign</td>
<td>1159</td>
</tr>
<tr>
<td>mask.array.fill</td>
<td>1159</td>
</tr>
<tr>
<td>mem.poly.allocator.class</td>
<td>647</td>
</tr>
<tr>
<td>mem.poly.allocator.ctor</td>
<td>648</td>
</tr>
<tr>
<td>mem.poly.allocator.eq</td>
<td>649</td>
</tr>
<tr>
<td>mem.poly.allocator.mem</td>
<td>648</td>
</tr>
<tr>
<td>mem.res</td>
<td>645</td>
</tr>
<tr>
<td>mem.res.class</td>
<td>646</td>
</tr>
<tr>
<td>mem.res.eq</td>
<td>647</td>
</tr>
<tr>
<td>mem.res.global</td>
<td>650</td>
</tr>
<tr>
<td>mem.res.monotonic.buffer</td>
<td>653</td>
</tr>
<tr>
<td>mem.res.monotonic.buffer.ctor</td>
<td>653</td>
</tr>
<tr>
<td>mem.res.monotonic.buffer.mem</td>
<td>654</td>
</tr>
<tr>
<td>mem.res.pool</td>
<td>650</td>
</tr>
<tr>
<td>mem.res.pool.ctor</td>
<td>652</td>
</tr>
<tr>
<td>mem.res.pool.mem</td>
<td>652</td>
</tr>
<tr>
<td>mem.res.private</td>
<td>651</td>
</tr>
<tr>
<td>mem.res.pool.overview</td>
<td>650</td>
</tr>
<tr>
<td>mem.res.syn</td>
<td>211</td>
</tr>
<tr>
<td>meta.alias</td>
<td>211</td>
</tr>
<tr>
<td>meta.constraints</td>
<td>464</td>
</tr>
<tr>
<td>meta.def</td>
<td>208</td>
</tr>
<tr>
<td>meta.future</td>
<td>210</td>
</tr>
<tr>
<td>meta.memdef</td>
<td>465</td>
</tr>
<tr>
<td>metaposix</td>
<td>465</td>
</tr>
<tr>
<td>meta.qual</td>
<td>43</td>
</tr>
<tr>
<td>meta.std</td>
<td>464</td>
</tr>
<tr>
<td>namespace.udir</td>
<td>214</td>
</tr>
<tr>
<td>numeric.iota</td>
<td>209</td>
</tr>
<tr>
<td>numeric.limits</td>
<td>1315</td>
</tr>
<tr>
<td>numeric.limits.members</td>
<td>481</td>
</tr>
<tr>
<td>numeric.ops</td>
<td>480</td>
</tr>
<tr>
<td>numeric.ops.gcd</td>
<td>1092</td>
</tr>
<tr>
<td>numeric.ops.lcm</td>
<td>1092</td>
</tr>
<tr>
<td>numeric.ops.midpoint</td>
<td>1092</td>
</tr>
<tr>
<td>numeric.ops.overview</td>
<td>1080</td>
</tr>
<tr>
<td>numeric.requirements</td>
<td>1094</td>
</tr>
<tr>
<td>numeric.special</td>
<td>485</td>
</tr>
<tr>
<td>numerics</td>
<td>459</td>
</tr>
<tr>
<td>numerics.defns</td>
<td>1142</td>
</tr>
<tr>
<td>numerics.general</td>
<td>1094</td>
</tr>
</tbody>
</table>
streambuf.virt.put (29.6.3.4.5) 1338
streambuf.virtuals (29.6.3.4) 1336
string.access (21.3.2.5) 731
string.accessors (21.3.2.7.1) 738
string.append (21.3.2.6.2) 732
string.assign (21.3.2.6.3) 733
string.capacity (21.3.2.4) 730
string.classes (21.3) 719
string.cmp (21.3.3.2) 741
string.compare (21.3.2.7.4) 739
string.cons (21.3.2.2) 728
string.conversions (21.3.4) 744
string.copy (21.3.2.6.7) 737
string.ends.with (21.3.2.7.6) 740
string.erase (21.3.2.6.5) 735
string.erasure (21.3.3.5) 743
string.find (21.3.2.7.2) 738
string.insert (21.3.2.6.4) 734
string.io (21.3.3.4) 742
string.iterators (21.3.2.3) 730
string.modifiers (21.3.2.6) 731
string.nonmembers (21.3.3) 740
string.op.append (21.3.2.6.1) 731
string.op.plus (21.3.3.1) 740
string.replace (21.3.2.6.6) 736
string.require (21.3.2.1) 727
string.special (21.3.3.3) 742
string.starts.with (21.3.2.7.5) 740
stringstreams (Clause 21) 714
stringstream.assign (21.3.2.6.7) 737
stringstream.cons (21.3.2.6.8) 737
structure (16.4.1) 447
structure.elements (16.4.1.1) 447
structure.requirements (16.4.1.3) 447
structure.specifications (16.4.1.4) 448
structure.summary (16.4.1.2) 447
support.contract (17.8) 498
support.contract.cviol (17.8.2) 498
support.coroutine (17.12) 511
support.dynamic (17.6) 489
support.exception (17.9) 499
support.general (17.1) 473
support.initlist (17.10) 502
support.initlist.access (17.10.3) 503
support.initlist.cons (17.10.2) 503
support.initlist.range (17.10.4) 503
support.limits (17.3) 477
support.limits.general (17.3.1) 477
support.rtti (17.7) 496
support.runtime (17.13) 515
support.signal (17.13.4) 516
support.start.term (17.5) 488
support.types (17.2) 473
support.types.byteops (17.2.5) 476
support.types.layout (17.2.4) 476
support.types.nullptr (17.2.3) 475
swappable.requirements (16.5.3.2) 458
syncstream (29.10) 1384
syncstream.osyncstream (29.10.3) 1387
syncstream.osyncstream.assign (29.10.3.3) 1388
syncstream.osyncstream.cons (29.10.3.2) 1387
syncstream.osyncstream.members (29.10.3.4) 1388
syncstream.osyncstream.overview (29.10.3.1) 1387
syncstream.syn (29.10.1) 1384
syncstream.syncbuf (29.10.2) 1384
syncstream.syncbuf.assign (29.10.2.3) 1385
syncstream.syncbuf.cons (29.10.2.2) 1385
syncstream.syncbuf.members (29.10.2.4) 1386
syncstream.syncbuf.overview (29.10.2.1) 1384
syncstream.syncbuf.special (29.10.2.6) 1386
synestream (29.10.2.5) 1386
syntax (4.3) 7
syserr (19.5) 535
syserr.compare (19.5.5) 541
syserr.errcat (19.5.2) 537
syserr.errcat.derived (19.5.2.4) 538
syserr.errcat.nonvirtuals (19.5.2.3) 538
syserr.errcat.objects (19.5.2.5) 538
syserr.errcat.overview (19.5.2.1) 537
syserr.errcat.virtuals (19.5.2.2) 537
syserr.errcode (19.5.3) 539
syserr.errcode.constructors (19.5.3.2) 539
syserr.errcode.modifiers (19.5.3.3) 539
syserr.errcode.nonmembers (19.5.3.5) 540
syserr.errcode.observers (19.5.3.4) 540
syserr.errcode.overview (19.5.3.1) 539
syserr.errcondition (19.5.4) 540
syserr.errcondition.constructors (19.5.4.2) 541
time.duration (27.5) 1194
time.duration.alg (27.5.9) 1200
time.duration.arithmetic (27.5.3) 1196
time.duration.cast (27.5.7) 1199
time.duration.comparisons (27.5.6) 1198
time.duration.cons (27.5.1) 1195
time.duration.io (27.5.10) 1200
time.duration.literals (27.5.8) 1199
time.duration.nonmember (27.5.5) 1197
time.duration.observer (27.5.2) 1196
time.duration.special (27.5.4) 1197
time.format (27.11) 1263
time.general (27.1) 1177
time.parse (27.12) 1266
time.point (27.6) 1201
time.point.arithmetic (27.6.3) 1202
time.point.cast (27.6.7) 1204
time.point.comparisons (27.6.6) 1203
time.point.cons (27.6.1) 1202
time.point.nonmember (27.6.5) 1203
time.point.observer (27.6.2) 1202
time.point.special (27.6.4) 1203
time.syn (27.2) 1177
time.tod (27.9) 1245
time.tod.hours (27.9.2) 1245
time.tod.io (27.9.6) 1248
time.tod.minutes (27.9.3) 1246
time.tod.overview (27.9.1) 1245
time.tod.seconds (27.9.4) 1247
time.tod.subsecond (27.9.5) 1247
time.traits (27.4) 1193
time.traits.duration.values (27.4.2) 1193
time.traits.is.clock (27.4.4) 1194
time.traits.is.fp (27.4.1) 1193
time.traits.specializations (27.4.3) 1194
time.zone (27.10) 1250
time.zone.db (27.10.2) 1250
time.zone.db.access (27.10.2.3) 1252
time.zone.db.list (27.10.2.2) 1251
time.zone.db.remote (27.10.2.4) 1252
time.zone.db.tzdb (27.10.2.1) 1250
time.zone.exception (27.10.3) 1252
time.zone.exception.ambig (27.10.3.2) 1253
time.zone.exception.nonexist (27.10.3.1) 1252
time.zone.general (27.10.1) 1250
time.zone.info (27.10.4) 1254
time.zone.info.local (27.10.4.2) 1255
time.zone.info.sys (27.10.4.1) 1254
time.zone.leap (27.10.8) 1261
time.zone.leap.members (27.10.8.2) 1262
time.zone.leap.nonmembers (27.10.8.3) 1262
time.zone.leap.overview (27.10.8.1) 1261
time.zone.link (27.10.9) 1263
time.zone.link.members (27.10.9.2) 1263
time.zone.link.nonmembers (27.10.9.3) 1263
time.zone.link.overview (27.10.9.1) 1263
time.zone.members (27.10.5.2) 1256
time.zone.nonmembers (27.10.5.3) 1256
time.zone.overview (27.10.5.1) 1255
time.zone.timezone (27.10.5) 1255
time.zone.zonedtime (27.10.7) 1257
time.zone.zonedtime.ctor (27.10.7.2) 1258
time.zone.zonedtime.members (27.10.7.3) 1260
time.zone.zonedtime.nonmembers (27.10.7.4) 1260
time.zone.zonedtime.overview (27.10.7.1) 1257
time.zone.zonedtimetraits (27.10.6) 1256
transformexclusive.scan (25.9.9) 1089
transform.inclusive.scan (25.9.10) 1090
tuple (20.5) 553
tuple.apply (20.5.3.5) 559
tuple.assign (20.5.3.2) 557
tuple.castr (20.5.3.1) 555
tuple.creation (20.5.3.4) 558
tuple.elem (20.5.3.7) 561
tuple.general (20.5.1) 553
tuple.helper (20.5.3.6) 560
tuple.rel (20.5.3.8) 561
tuple.special (20.5.3.10) 562
tuple.swap (20.5.3.3) 558
tuple.syn (20.5.2) 553
tuple.trait (20.5.3.9) 562
tuple.tuple (20.5.3) 554
type.descriptions (16.4.2.2) 450
type.descriptions.general (16.4.2.2.1) 450
type.index (20.17) 708
type.index.hash (20.17.4) 709
type.index.members (20.17.3) 708
type.index.overview (20.17.2) 708
type.index.synopsis (20.17.1) 708
type.info (17.7.2) 497
typeinfo.syn (17.7.1) 496
uncaught.exceptions (17.9.5) 501
underflow.error (19.2.10) 533
uninitialized.construct.default (20.10.11.3) 613
uninitialized.construct.value (20.10.11.4) 614
uninitialized.copy (20.10.11.5) 614
uninitialized.fill (20.10.11.7) 616
uninitialized.move (20.10.11.6) 615
unique.ptr (20.11.1) 618
unique.ptr.create (20.11.1.4) 625
unique.ptr.dltr (20.11.1.1) 618
unique.ptr.dltr.dflt (20.11.1.1.2) 619
unique.ptr.dltr.dflt1 (20.11.1.1.3) 619
unique.ptr.dltr.general (20.11.1.1.1) 618
unique.ptr.dltr.ctor (20.11.1.3.1) 627
unique.ptr.runtime (20.11.1.3) 623
unique.ptr.runtime.asgn (20.11.1.3.2) 625
unique.ptr.runtime.ctor (20.11.1.3.1) 624
unique.ptr.runtime.members (20.11.1.3.4) 625
unique.ptr.runtime.modifiers (20.11.1.3.3) 625
unique.ptr.single (20.11.1.2) 619
unique.ptr.single.asgn (20.11.1.2.3) 622
unique.ptr.single.ctor (20.11.1.2.1) 620
unique.ptr.single.dtor (20.11.1.2.2) 622
unique.ptr.single.modifiers (20.11.1.2.5) 623

Cross references 1634
view.interface.members (24.5.2.1) 947
views (22.7) 872
views.general (22.7.1) 872
views.span (22.7.3) 872
wide.stream.objects (29.4.4) 1315
zombie.names (16.5.4.3.1) 465
Cross references from ISO C++ 2017

All clause and subclause labels from ISO C++ 2017 (ISO/IEC 14882:2017, Programming Languages — C++) are present in this document, with the exceptions described below.

alg.all_of see alg.all_of
alg.any_of see alg.any.of
alg.is_permutation see alg.is_permutation
alg.none_of see alg.none.of
any.bad_any_cast see any.bad.any.cast
array.data see array.members
array.fill see array.members
array.size see array.members
array.swap see array.members
back.insert.iter.cons see back.insert.iter.ops
back.insert.iter.op* see back.insert.iter.ops
back.insert.iter.op++ see back.insert.iter.ops
back.insert.iter.op= see back.insert.iter.ops
basic.scope.proto see basic.scope.param
char.traits.specializations.char16.t see char.traits.specializations.char16.t
char.traits.specializations.char32.t see char.traits.specializations.char32.t
char.traits.specializations.char8.t see char.traits.specializations.char8.t
class.copy see class.mem
class.mfct.non-static see class.mfct.non-static
comparisons.equal_to see comparisons.equal.to
comparisons.greater_equal see comparisons.greater.equal
comparisons.less_equal see comparisons.less.equal
comparisons.not_equal_to see comparisons.not_equal_to
condition_variable.syn see condition_variable.syn
depr.complex.syn see depr.complex.h.syn
depr.cpp.headers removed
depr.stdalign.syn see depr.stdalign.h.syn
depr.stdbool.syn see depr.stdbool.h.syn
depr.ctgmath.syn see depr.tgmath.h.syn
depr.default.allocator removed
depr.except.spec removed
depr.func.adaptor.binding removed
depr.func.adaptor.typedefs removed
depr.negators removed
depr.static_constexpr see depr.static.constexpr
depr.storage.iterator removed
depr.uncaught removed
depr.util.smartptr.shared.obs removed
depr.weak.result_type removed
deque.special removed
diff.conv see diff.expr
diff.cpp03.conv see diff.cpp03.expr
diff.cpp03.dcl.dcl see diff.cpp03.dcl.dcl
diff.cpp03.special see diff.cpp03.class
diff.cpp11.dcl.dcl see diff.cpp11.dcl.dcl
diff.cpp14.dcl see diff.cpp14.dcl.dcl
diff.cpp14.special see diff.cpp14.class
diff.cpp17.dcl.dcl see diff.cpp17.dcl.dcl
diff.cpp17.special see diff.cpp17.class
diff.special see diff.class
expr.pseudo see expr.prim.id.dtor
facets.examples removed
forward_list.erase see forward_list.erase
forward_list.syn see forward_list.syn
forward_list.special removed
front.insert.iter.cons see front.insert.iter.ops
front.insert.iter.op* see front.insert.iter.ops
front.insert.iter.op++ see front.insert.iter.ops
front.insert.iter.op= see front.insert.iter.ops
fs.class.directory_iterator see fs.class.directory_iterator
fs.class.directory_status see fs.class.directory_status
fs.class.filesystem_error see fs.class.filesystem_error
fs.class.filesystem_error_members see fs.class.filesystem_error_members
fs.OP.copy_file see fs.OP.copy_file
fs.OP.copy_syalnk see fs.OP.copy_symlink
fs.OP.create_dir_syalnk see fs.OP.create_dir_symlink
fs.OP.create_directories see fs.OP.create_directories
fs.OP.create_directory see fs.OP.create_directory
fs.OP.create_hardlk see fs.OP.create_hardlk
fs.OP.create_syalnk see fs.OP.create_symlink
fs.OP.current_path see fs.OP.current_path
fs.OP.file_size see fs.OP.file.size
fs.OP.hardlk_ct see fs.OP.hardlk_ct
fs.OP.is_block_file see fs.OP.is_block_file
fs.OP.is_char_file see fs.OP.is_char_file
fs.OP.is_directory see fs.OP.is_directory
fs.OP.is_empty see fs.OP.is_empty
time.traits.duration_values see time.traits.duration.values

time.traits.is_clock see time.traits.is.clock
time.traits.is_fp see time.traits.is.fp

unord.map.swap removed
unord.multimap.swap removed
unord.multiset.swap removed
unord.set.swap removed
util.smartptr removed
util.smartptr.shared.atomic see depr.util.smartptr.shared.atomic
utility.as_const see utility.as_const
utility.from.chars see charconv.from.chars
utility.to.chars see charconv.to.chars

variant.traits removed
vector.special removed
Index

!, see operator, logical negation
!, see operator, inequality
(), see operator, function call, see declarator, function
*, see operator, indirection, see operator, multiplication, see declarator, pointer
+, see operator, unary plus, see operator, addition
++, see operator, increment
,, see operator, comma
->, see operator, class member access
->*, see operator, pointer to member
--, see operator, decrement
., see operator, class member access
.*, see operator, pointer to member
., see operator, comma
-, see operator, unary minus, see operator, subtraction
->, see operator, class member access
->*, see operator, pointer to member
->*, see operator, comma
, see operator, comma

\, see backslash character
/, see operator, division

?:, see operator, conditional expression
[]], see operator, subscripting, see declarator, array

\# operator, 433, 435
\## operator, 436
#define, 433
#define, 429
else, 430
#endif, 430
#error, see preprocessing directives, error
#if, 429, 470
ifdef, 430
ifndef, 430
#include, 431, 456
#line, see preprocessing directives, line control
#pragma, see preprocessing directives, pragma
#undef, 436, 467
%, see operator, remainder
&, see operator, address-of, see operator, bitwise
AND, see declarator, reference

&&, see operator, logical AND
~, see operator, bitwise negate OR
\, see backslash character

/Cppl7CopyInsertable into X, 764
/Cppl7DefaultInsertable into X, 764
/Cppl7EmplaceConstructible into X from args, 764
/Cppl7Erasable from X, 765
/Cppl7MoveInsertable into X, 764
{
block statement, 142
class declaration, 242
class definition, 242
dependency graph, 204
initializer list, 187

_, see character, underscore

__<cplusplus, 439
__DATE, 439
__FILE, 439
__LINE, 439

__STDCPP_DEFAULT_NEW_ALIGNMENT__, 440
__STDCPP_STRONG_POINTER_SAFETY__, 441
__STDCPP_THREADS__, 441
__STDC_ISO_10646__, 441
__STDC_MB_MIGHT_NEQ_WC__, 441
__STDC_VERSION__, 441
__STDC__, 441
__TIME__, 440
__VA_ARGS__, 433, 434
__VA_OPT__, 433, 434
__cpp_aggregate_bases, 440
__cpp_aggregate_nsmdi, 440
__cpp_aggregate_paren_init, 440
__cpp_alias_templates, 440
__cpp_aligned_new, 440
__cpp_attributes, 440
__cpp_binary_literals, 440
__cpp_capture_star_this, 440
__cpp_conditional_explicit, 440
__cpp_constexpr, 440
__cpp_coroutines, 440
__cpp_decltype, 440
__cpp_decltype_auto, 440
__cpp_deduction_guides, 440
__cpp_guaranteed_copy_elision, 440
__cpp_hex_float, 440
__cpp_impl_destroying_delete, 440
Index

abbreviated
  template function, see template, function, abbreviated

abort, 80, 147
absolute path, 1394
abstract class, see class, abstract
abstract-declarator, 170, 1557
abstract-pack-declarator, 170, 1557
access, 3
access control, 280–289
  anonymous union, 268
  base class, 283
  base class member, 270
  class member, 108
default, 280
default argument, 282
friend function, 285
member function and, 252
member name, 280
multiple access, 288

nested class, 289
overload resolution and, 278
overloading and, 307
private, 280
protected, 280, 287
public, 280
using-declaration and, 218
virtual function, 288
access specifier, 282, 283
access-specifier, 270, 1561
accessible, 284
active
  union member, 267
addition operator, see operator, addition
additive-expression, 127, 1552
address, 67, 130
addressable function, 465
aggregate, 187
elements, 187
aggregate initialization, 187
algorithm
  stable, 447, 470
<algorithm>, 991
alias
  namespace, 211
alias template, 369
alias-declaration, 151, 1555
alignas, 15, 222, 223, 1559
alignment, 59
  extended, 60
  fundamental, 60
  new-extended, 60
  stricter, 60
  stronger, 60
  weaker, 60
alignment requirement
  implementation-defined, 59
alignment-specifier, 222, 1559
alignof, 15, 115, 125, 381, 382, 1551
aligned type, see type, allocated
allocation
  alignment storage, 121
  implementation-defined bit-field, 265
  unspecified, 248
allocation functions, 57
alternative token, see token, alternative
ambiguity
  base class member, 278
  class conversion, 280
  declaration type, 153
declaration versus cast, 171
declaration versus expression, 149
function declaration, 184
member access, 278
overloaded function, 307
parentheses and, 119
ambiguous conversion sequence, see conversion sequence, ambiguous
Amendment 1, 467
Index

array, 796
array
as aggregate, 799
contiguous storage, 799
initialization, 799, 800
tuple interface to, 801
zero sized, 800
array size
default, 175

arrow operator, see operator, class member access
as-if rule, 6
asm, 15, 219, 1559
implementation-defined, 219
asm-definition, 219, 1559
assembler, 219
assert, 225, 1560
<assert.h>, 456, 533, 1592
assertion, 226
assignment
and lvalue, 134
copy, see assignment operator, copy
move, see assignment operator, move, 445
reference, 191
assignment operator
copy, 251, 256–258
hidden, 257
implicitly declared, 256
implicitly defined, 257
non-trivial, 257
trivial, 257
virtual bases and, 258
move, 251, 256–258
hidden, 257
implicitly declared, 257
implicitly defined, 257
non-trivial, 257
trivial, 257
overloaded, 329
assignment-expression, 134, 1553
assignment-operator, 134, 1553
associated constraints, 349
associative containers
exception safety, 782
requirements, 782
unordered, see unordered associative containers
asynchronous provider, 1533
asynchronous return object, 1533
at least as constrained, 351
at least as specialized as, see more specialized
atexit, 80
<atomic>, 1477
atomic constraint, 348
identical, 348
atomic operations, see operation, atomic
atomic smart pointers, 642–645
attached
declaration, 233
type, 50
attribute, 222–225
alignment, 223
carries dependency, 224
contracts, 225–228
deprecated, 229
fallthrough, 229
likely, 229
maybe unused, 230
no unique address, 231
nordiscard, 230
noreturn, 231
syntax and semantics, 222
unlikely, 229
attribute, 222, 1559
attribute-argument-clause, 222, 1560
attribute-declaration, 151, 1555
attribute-list, 222, 1559
attribute-namespace, 222, 1560
attribute-scoped-token, 222, 1560
attribute-specifier, 222, 1559
attribute-specifier-seq, 222, 1559
attribute-token, 222, 1560
attribute-using-prefix, 222, 1559
audit, 14, 225, 1560
auto, 15, 146, 165, 1556
automatic storage duration, see storage duration, automatic
await-expression, 117, 1552
awk, 1447
axiom, 14, 225, 1560
backslash character, 18
bad_alloc, 122
bad_cast, 110
bad_typeid, 111
balanced-token, 222, 1560
balanced-token-seq, 222, 1560
base characteristic, 681
base class, 269–271
dependent, 378
direct, 270
indirect, 270
non-virtual, 271
overloading and, 306
private, 283
protected, 283
public, 283
virtual, 271
base class subobject, 51
base-2 representation, 66
base-clause, 269, 1561
base-specifier, 269, 1561
base-specifier-list, 269, 1561
behavior
conditionally-supported, 3, 6
default, 445, 449
implementation-defined, 4, 7
locale-specific, 4
observable, 6, 7
on receipt of signal, 71
required, 446, 449
undefined, 5–7, 931
unspecified, 5, 7
Ben, 306
Bernoulli distributions, 1127–1130
bernoulli_distribution
discrete probability function, 1127
Bessel functions
I_ν, 1172
J_ν, 1172
K_ν, 1173
N_ν, 1173
j_n, 1175
n_n, 1175
beta functions B, 1171
better conversion, see conversion, better
better conversion sequence, see conversion sequence, better
binary fold, 102
binary left fold, 102
binary operator
interpretation of, 329
overloaded, 329
binary right fold, 102
binary exponent-part, 19, 1548
binary-literal, 15, 1547
bind directly, 193
binding
reference, 192
binomial_distribution
discrete probability function, 1127
<bit>, 1103
bit-field, 265
address of, 265
alignment of, 265
implementation-defined alignment of, 265
implementation-defined sign of, 1575
type of, 265
unnamed, 265
zero width of, 265
bitand, 15
bitmask
element, 451
empty, 451
value
clear, 451
is set, 451
set, 451
bitor, 15
<bitset>, 591
block, 3
initialization in, 148
block scope, 32
block statement, see statement, compound
block structure, 148
block with forward progress guarantee delegation, 76
block-declaration, 151, 1555
body
function, 198
Bond
James Bond, 99
bool, 15, 162, 1556
Boolean literal, 22
boolean literal, see literal, boolean
Boolean type, 66  
boolean-literal, 22, 1549  
bound argument entity, 661  
bound arguments, 674  
bound, of array, 175  
brace-or-equal-initializer, 183, 1558  
braced-init-list, 183, 1558  
brains  
  names that want to eat your, 465  
break, 15  
break ;, 147, 1554  
buckets, 783  
built level, 228  
built-in candidate, 311  
built-in operators, see operators, built-in  
byte, 50, 118

C  
  linkage to, 220  
  standard, 1  
  standard library, 2  
c-char, 17, 1547  
c-char-sequence, 17, 1547  
call  
  operator function, 328  
call pattern, 662  
call signature, 661  
call wrapper, 661  
    forwarding, 661  
    perfect forwarding, 662  
    simple, 661  
    type, 661

callable object, see object, callable  
callable type, see type, callable, 676  
candidate function, 307  
capture  
  implicit, 99  
capture, 98, 1550  
capture-default, 97, 1550  
capture-list, 98, 1550  
captured, 99  
    by copy, 100  
    by reference, 101  
carry  
    carries a dependency, 73  
carry  
    subtract_with_carry_engine, 1117

case, 15, 142, 144, 1554  
<cassert>, 456, 533, 1309, 1592  
cast  
  base class, 112  
  const, 114, 125  
  derived class, 112  
  dynamic, 109, 497  
    construction and, 298  
    destruction and, 298  
  integer to pointer, 113  
  lvalue, 111, 113  
  pointer to integer, 113  
  pointer-to-function, 113  
  pointer-to-member, 113, 114  
  reference, 111, 114  
  reinterpret, 113, 125  
  integer to pointer, 113  
  lvalue, 113  
  pointer to integer, 113  
  pointer-to-function, 113  
  pointer-to-member, 114  
  reference, 114  
  static, 111, 125  
  lvalue, 111  
  reference, 111
  undefined pointer-to-function, 113

cast-expression, 125, 1552  
casting, 108  
casting away constness, 115  
catch, 15, 418  
cats  
    interfering with canines, 496  
cauchy_distribution  
    probability density function, 1136

<cclass>, 755  
<cerrno>, 467, 533  
<cfenv>, 1094  
char, 15, 162, 1556  
    implementation-defined sign of, 66  
char-like object, 714  
char-like type, 714  
char16_t, 15, see type, char16_t, 162, 1556  
char32_t, 15, see type, char32_t, 162, 1556  
char8_t, 15, see type, char8_t, 162, 1556  
char_class_type  
    regular expression traits, 1439  
character, 444  
    decimal-point, 451  
    multibyte, 4  
    null, 10  
    signed, 66  
    source file, 9  
    terminating null, 452  
    underscore, 14
    in identifier, 14  
character literal, see literal, character  
character sequence, 451  
character set, 10–11  
    basic execution, 10, 50  
    basic source, 9, 10  
    execution, 11
character string, 20  
character string literal, 435  
character-literal, 17, 1547  
<charconv>, 711  
checking  
    point of error, 372  
    syntax, 372
chi_squared_distribution  
    probability density function, 1135
<chrono>, 1177  
chunks, 650  

Index 1644
Index 1645

#include <cinttypes>, 1435, 1436
class, 67, 242–266
  abstract, 277
  base, 467, 471
cast to incomplete, 126
class constructor and abstract, 277
definition, 26
derived, 471
linkage of, 48
linkage specification, 220
local, 269, see local class
member function, see member function, class
nested, 265, see nested class
polymorphic, 272
scope of enumerator, 207
standard-layout, 64, 243
trivial, 64, 243
trivially copyable, 64, 243
union-like, 268
unnamed, 157
variant member of, 268
class, 15, 205, 242, 337, 1558, 1561, 1563
class name
  elaborated, 162, 245
  point of declaration, 245
  scope of, 244
typedef, 156, 157, 245
class object
  assignment to, 134
  member, 248
sizeof, 118
class object copy, see constructor, copy
class object initialization, see constructor
class-head, 242, 1560
class-head-name, 242, 1560
class-key, 242, 1561
class-name, 242, 1560
class-or-decltype, 270, 1561
class-specifier, 242, 1560
class-virt-specifier, 242, 1561
#include <climits>, 1598
#include <clocale>, 451, 1592
closure object, 94
closure type, 94
#include <cmath>, 1161, 1170
co_await, 15, 116
co_return, 15, 148
co_yield, 15, 133
#include <codecvt>, 1607
coherence
  read-read, 74
  read-write, 74
  write-read, 74
  write-write, 74
coherence-ordered before, 1481
collating element, 1438
comma operator, see operator, comma
comment, 11–12
  /* */ , 12
  // , 12
common comparison type, 301
common initial sequence, 248
#include <compare>, 129, 503
compare-expression, 128, 1552
correspondence
  pointer, 130
  pointer to function, 130
  undefined pointer, 127
correspondence category types, 504
compilation
  separate, 9
compiler control line, see preprocessing directive
compl, 15
complete object, 51
complete object of, 52
complete-class context, 247
completely defined, 247
#include <complex>, 1095
#include <complex.h>, 1591, 1595
component, 444
composite pointer type, 84
compound requirement, 104, 1551
compound statement, 142, 1554
concatenation
  macro argument, see ## operator
  string, 21
concept, 370
type, 370
concept, 15, 335, 1562
concept-definition, 335, 1562
concept-name, 335, 1562
conditional forward progress guarantees, 76
condition, 141, 1553
conditions
  rules for, 141
#include <condition_variable>, 1525
conditional-expression
  throw-expression in, 132
conditional-expression, 132, 1553
conditionally-supported behavior, see behavior, conditionally-supported
conditionally-supported-directive, 427, 1564
conflict, 72
conformance requirements, 6–7
class templates, 6
classes, 6
general, 6
library, 6
method of description, 6
conjunction, 348
consistency
  linkage, 154
  linkage specification, 221
type declaration, 50
const, 15, 68, 170, 1557
  cast away, 115
constructor and, 251, 253
destructor and, 251, 258
linkage of, 48
overloading and, 305
const member function, 251
const object, see object, const
undefined change to, 161
const volatile member function, 251
const volatile object, see object, const volatile
const-default-constructible, 184
const-qualified, 68
const-volatile-qualified, 68
const_cast, 15, see cast, const, 106, 381, 382, 1551
const_local_iterator, 785
constant, 15, 90
enumeration, 205
null pointer, 88, 89
constant expression, 135, see expression, constant permitted result of, 138
constant initialization, 77
constant initializer, 135
constant iterator, 885
constant subexpression, 444
constant-expression, 135, 1553
constexpr, 15, 153, 1555
constexpr, 15, 142, 153, 1554, 1555
constexpr function, 157
constexpr if, 143
constexpr iterators, 886
constituent expression, 69
constraint, 348
associated, see associated constraints
immediately-declared, 336
normalization, 350
satisfaction
atomic, 348
conjunction, 348
disjunction, 348
subsumption, 350
constraint-expression, 349, 1563
constraint-logical-and-expression, 335, 1562
constraint-logical-or-expression, 335, 1562
construction, 296–298
dynamic cast and, 298
member access, 296
move, 445
pointer to member or base, 297
typeid operator, 297
virtual function call, 297
constructor, 252
address of, 253
array of class objects and, 290
converting, 261
copy, 61, 251, 254–256, 452
elision, 298
implicitly declared, 255
implicitly defined, 255
nontrivial, 255
trivial, 255
default, 251, 253
non-trivial, 253
trivial, 253
exception handling, see exception handling, constructors and destructors
explicit call, 252
implicitly called, 253
implicitly defined, 253
inheritance of, 253
inherited, 214
move, 251, 254–256
elision, 298
implicitly declared, 255
implicitly defined, 255
non-trivial, 255
trivial, 255
non-trivial, 253
random number distribution requirement, 1113
random number engine requirement, 1110
union, 267
constructor, conversion by, see conversion, user-defined
contained value
any, 587
optional, 565
variant, 577
container
contiguous, 764
contains a value
optional, 565
context
non-deduced, 410
contextually converted constant expression of type
bool, see conversion, contextual
contextually converted to bool, see conversion, contextual
contextually implicitly converted, 85
contiguous container, 764
continue, 15
and handler, 418
and try block, 418
continue ;, 147, 1554
contract, 226
predicate, 226
contract condition, 226
contract violation, 228
contract-attribute-specifier, 225, 1560
contract-level, 225, 1560
control line, see preprocessing directive
control-line, 427, 1564
conventions, 449
lexical, 9–23
conversion
argument, 177
array-to-pointer, 86
better, 324
bool, 88
usual, 58
decallocation functions, 57
decay
  array, see conversion, array-to-pointer
  function, see conversion, function-to-pointer
decimal floating literal, see literal, decimal floating
decimal-floating-literal, 19, 1548
decimal-literal, 15, 1547
decl-reachable, 237
decl-specifier, 153, 1555
decl-specifier-seq, 153, 1555
declaration, 24, 151–225
  array, 175
  asm, 219
  bit-field, 265
  class name, 25
  constant pointer, 172
default argument, 180–182
definition versus, 24
  ellipsis in function, 108, 177
eumerator point of, 31
exported, 233
extern, 24
external reference, 191
  forward, 154
  forward class, 245
  function, 24, 152, 176
  local class, 269
  member, 245
  multiple, 50
  name, 24
  object, 152
  opaque enum, 25
  overloaded, 304
overloaded name and friend, 286
parameter, 25, 177
parentheses in, 171, 172
pointer, 172
reference, 173
  static member, 25
  storage class, 153
  structured binding, see structured binding
    declaration
type, 172
typedef, 152
typedef as type, 25
declaration, 151, 1554
declaration hiding, see name hiding
declaration-seq, 151, 1554
declarative region, 30
declarator, 25, 152, 168–198
  array, 175
  function, 176–180
  meaning of, 172–182
  multidimensional array, 175
  pointer, 172
  pointer-to-member, 174
reference, 173
declarator, 160, 1556
declarator-id, 170, 1557
decltype, 15, 164, 165, 1556
decltype-specifier, 164, 1556
decrement operator
  overloaded, see overloading, decrement operator
deduction
  class template argument, 417
class template arguments, 108, 162, 168, 314
  placeholder type, 167
deduction-guide, 417, 1563
default, 15, 142, 198, 225, 1554, 1558, 1560
default access control, see access control, default
default argument
  overload resolution and, 316
default argument instantiation, 390
default constructor, see constructor, default
  random number distribution requirement, 1112
  seed sequence requirement, 1108
default initializers
  overloading and, 306
default member initializer, 247
default memory resource pointer, 650
default-initialization, 183
default-inserted, 764
defaulted, 200
deferred function, 1541
define, 24
defined, 428
defined-macro-expression, 428, 1564
defining-type-id, 170, 1557
defining-type-specifier, 160, 1556
defining-type-specifier-seq, 160, 1556
definition, 24, 25
  alternate, 467
  class, 242, 246
  class name as type, 244
  constructor, 198
coroutine, 201
declaration as, 152
  function, 198–201
deleted, 200
explicitly-defaulted, 199
local class, 269
member function, 249
namespace, 208
nested class, 265
program semantics affected by, 389
pure virtual function, 277
scope of class, 244
static member, 264
virtual function, 275
definitions, 3–5
delete
  array, 123
  single-object, 123
delete, 15, 57, 123, 198, 302, 381, 1552, 1558
destructor and, 123, 259
operator
replaceable, 468
overloading and, 58
single-object, 123
type of, 302
undefined, 123
delete-expression, 123, 1552
deleter, 618
denormalized value, see number, subnormal
dependency-ordered before, 73
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>, 796
dereferencing, see indirection
derivation, see inheritance
derived class, 269–277
most, see most derived class
derived object
most, see most derived object
designated-initializer-clause, 183, 1558
designated-initializer-list, 183, 1558
designator, 183, 1558
deztringization, 442
destroying operator delete, see operator delete,
destroying
destruction, 296–298
dynamic cast and, 298
member access, 296
pointer to member or base, 297
typeid operator, 297
virtual function call, 297
destructor, 258, 452
default, 258
exception handling, see exception handling,
constructors and destructors
explicit call, 259
implicit call, 259
implicitly defined, 259
non-trivial, 259
program termination and, 259
pure virtual, 259
union, 267
virtual, 259
diagonsable rules, 6
diagnostic message, see message, diagnostic
digit, 13, 1546
digit-sequence, 19, 1548
digraph, see token, alternative, 12
direct base class, see base class, direct
direct member, 246
direct-initialization, 185
direct-list-initialization, 193
direct-non-list-initialization, 445
directed acyclic graph, see DAG
copy, see constructor, copy, elision

copy constructor, see constructor, copy, elision

move constructor, see constructor, move, elision

ellipsis

conversion sequence, 108, 321

overload resolution and, 316

elliptic integrals

complete \( \Pi \), 1172

complete \( E \), 1172

complete \( K \), 1171

incomplete \( \Pi \), 1174

incomplete \( E \), 1173

incomplete \( F \), 1173

else, 15, 142, 143, 1554

else-group, 427, 1564

empty

future object, 1537

empty shared_future object, 1539

empty-declaration, 151, 1555

enclosing namespace set, 209

enclosing statement, 141

encoding

multibyte, 21

encoding-prefix, 17, 1547

don-of-file, 596

def-line, 427, 1564

eengine, see random number engine

eengine adaptor, see random number engine

eadaptor

eengines with predefined parameters

default_random_engine, 1122

knuth_b, 1122

minstd_rand, 1121

minstd_rand0, 1121

mt19937, 1121

mt19937_64, 1121

ranlux24, 1122

ranlux24_base, 1122

ranlux48, 1122

ranlux48_base, 1122

ensures, 225, 1560

entity, 24

associated, 39

local, 24

templated, 336

enum, 15, 67, 162, 205, 1556, 1558

overloading and, 305

type of, 204, 206

underlying type, see type, underlying

eenum name

typedef, 157

enum-base, 205, 1558

enum-head, 205, 1558

enum-head-name, 205, 1558

enum-key, 205, 1558

enum-name, 204, 1558

enum-specifier, 205, 1558

enumerated element, 450

enumerated type, see type, enumerated
enumeration, 204

linkage of, 48

scoped, 205

unscoped, 205

enumeration scope, 34

tenumeration type

cconversion to, 112

static_cast

cconversion to, 112
tenumerator

definition, 26

scoped, 205

unscoped, 205

tvalue of, 205
tenumerator, 205, 1558
tenumerator-definition, 205, 1558
tenumerator-list, 205, 1558
tenvironment

program, 77

epoch, 1192
tequality

strong structural, 300
tequality-expression, 130, 1553
equivalence
template type, 351
type, 155, 244
equivalent

texpressions, 366

function templates, 367

functionally, see functionally equivalent
template-heads, 366
template-parameters, 366
equivalent parameter declarations, 305
tonoverloading and, 305
equivalent-key group, 783
equivalently-valued, 463
<errno.h>, 533
tescape character, see backslash character
tescape sequence

undefined, 18
tescape-sequence, 17, 1547
tEulerian integral of the first kind, see beta
tevaluation, 71	onder of argument, 107
tsignal-safe, 517
	nunspecified order of, 71, 78
	nunspecified order of argument, 107
	nunspecified order of function call, 107
texception

tarithmetic, 81

undefined arithmetic, 81
<exception>, 499
texception handling, 418–426
tconstructors and destructors, 420
tcurrently handled exception, 422
texception object, 419, 420
constructor, 420
destructor, 420
function try block, 418
goto, 418
handler, 418, 419, 421–422, 471
active, 422
array in, 421
incomplete type in, 421
match, 421–422
pointer to function in, 421
rvalue reference in, 421
memory, 420
nearest handler, 419
rethrow, 134, 420
rethrowing, 420
switch, 418
terminate called, 134, 420, 423
throwing, 134, 419
try block, 418
exception object, see exception handling, exception object
exception specification, 422–425
noexcept
constant expression and, 423
non-throwing, 423
potentially-throwing, 422
virtual function and, 423
exception-declaration, 418, 1563
exclusive-or-expression, 131, 1553
<execution>, 709
execution agent, 1501
execution policy, 709
execution step, 75
exit, 77, 79, 147
expects, 225, 1560
explicit, 15, 155, 417, 1555, 1563
explicit type conversion, see casting
explicit-instantiation, 392, 1563
explicit-specialization, 394, 1563
explicit-specifier, 155, 1555
explicitly captured, 98
explicitly initialized elements
aggregate, 187
exponent-part, 19, 1548
exponential integrals Ei, 1174
exponential_distribution
probability density function, 1130
export, 15, 232, 233, 236, 1560
export-declaration, 233, 1560
expr-or-braced-init-list, 183, 1558
expression, 81–140
additive operators, 127
alignof, 125
assignment and compound assignment, 134
await, 116
bitwise AND, 131
bitwise exclusive OR, 131
bitwise inclusive OR, 131
cast, 108, 125–126
class member access, 108
comma, 135
conditional operator, 132
const cast, 114
constant, 135, 138
converted constant, 138
core constant, 136
decrement, 109, 116
delete, 123
destructor call, 93
dynamic cast, 109
equality operators, 130
equality-preserving, 518
equivalent, see equivalent, expressions
fold, 102
function call, 106
functionally equivalent, see functionally equivalent, expressions
increment, 109, 116
integral constant, 137
lambda, 93–102
left-shift-operator, 128
logical AND, 131
logical OR, 131
multiplicative operators, 127
new, 119
noexcept, 125
order of evaluation of, 81
parenthesized, 91
pointer-to-member, 126
pointer-to-member constant, 115
postfix, 105–115
potentially constant evaluated, 139
potentially evaluated, 26
primary, 90–105
pseudo-destructor call, 93
reference, 83
reinterpret cast, 113
relational operators, 129
requires, 102–105
right-shift-operator, 128
rvalue reference, 82
sizeof, 118
spaceship, 128
static cast, 111
three-way comparison, 128
throw, 134
type identification, 111
type-dependent, 376
unary, 115–125
unary operator, 115
value-dependent, 376
yield, 133
expression, 135, 1553
expression-equivalent, 445
expression-list, 106, 1551
expression-statement, 142, 1554
extend, see namespace, extend
extended alignment, 60
extended integer type, 66
extended signed integer type, 65
extended unsigned integer type, 65
extern, 15, 153, 220, 392, 1555, 1559, 1563
linkage of, 154
extern "C", 456, 467
extern "C++", 456, 467
extern template, see instantiation, explicit external linkage, 48
extreme_value_distribution
probability density function, 1133
F (incomplete elliptic integrals), 1173
facet, 1275
fallback-separator, 1397
false, 15
file, 1389
file attributes, 1414
cached, 1414
file system, 1389
file system race, 1390
file, source, see source file
filename, 1397
filename, 1397
<filesystem>, 1390
final, 14, 242, 246, 1561
final overrider, 273
final suspend point, 202
finite state machine, 1438
fisher_f_distribution
probability density function, 1136
float, 15, 162, 1556
floating literal, see literal, floating
floating-literal, 19, 1548
floating-point literal, see literal, floating
floating-point promotion, 88
floating-point type, see type, floating-point
implementation-defined, 66
floating-suffix, 19, 1548
fold
binary, 102
unary, 102
fold-expression, 102, 1550
fold-operator, 102, 1550
for, 15, 144–146, 1554
scope of declaration in, 146
for-range-declaration, 144, 1554
for-range-initializer, 144, 1554
format specifier, 1438
forward, 547
forward progress guarantees
concurrent, 76
delegation of, 76
parallel, 76
weakly parallel, 76
<forward_list>, 797
forwarding reference, 406
fractional-constant, 19, 1548
free store, see also new, delete, 302
freestanding implementation, 6
friend
virtual and, 275
access specifier and, 286
class access and, 285
inheritance and, 286
local class and, 287
template and, 359
friend, 15, 153, 162, 286, 1555
friend function
access and, 285
inline, 286
linkage of, 286
member function and, 285
nested class, 266
<fstream>, 1371
full-expression, 70
function, see also friend function; member function; inline function; virtual function allocation, 57, 120
comparison, 444
conversion, 262
deallocation, 58, 302
definition, 26
global, 467, 470
handler, 445
handler of type, 421
inline, 159
linkage specification overloaded, 221
modifier, 445
named by expression or conversion, 26
needed for constant evaluation, 140
observer, 445
operator, 328
template, 328
overload resolution and, 308
overloaded, see overloading
overloading and pointer versus, 305
parameter of type, 177
pointer to member, 126
program semantics affected by the existence of a function definition, 389
replacement, 446
reserved, 446
template parameter of type, 338
viable, 307
virtual, 272–276
pure, 277
virtual function call, 106
virtual member, 467
function argument, see argument function call, 107
recursive, 108
undefined, 113
function call operator overloaded, 330
function object, 658
binders, 672–674
mem_fn, 674
Index
Index

reference_wrapper, 662
  type, 658
  wrapper, 674–678
  function parameter, see parameter
  function parameter pack, 356
  function parameter scope, 32
  function pointer type, 67
  function return, see return
  function return type, see return type
  function try block, see exception handling,
    function try block
  function-body, 198, 1558
  function-definition, 198, 1558
  function-like macro, see macro, function-like
  function-local predefined variable, see variable,
    function-local predefined
  function-specifier, 155, 1555
  function-try-block, 418, 1563
  <functional>, 658
  functionally equivalent
    expressions, 366
    function templates, 367
    template-heads, 367
  functions
    candidate, 383
    fundamental alignment, 60
    fundamental type, 67
    destructor and, 260
    fundamental type conversion, see conversion,
      user-defined
    future
    shared state, 1533
  <future>, 1531

gamma_distribution
  probability density function, 1131

generate
  seed sequence requirement, 1108
  generated destructor, see destructor, default
  generation algorithm
    discard_block_engine, 1119
    independent_bits_engine, 1120
    linear_congruential_engine, 1115
    mersenne_twister_engine, 1116
    shuffle_order_engine, 1120
    subtract_with_carry_engine, 1117
  generic lambda, 94
  geometric_distribution
    discrete probability function, 1128
  global module, see module, global
  global module fragment, 237
  global name, see name, global
  global namespace, see namespace, global
  global namespace scope, see namespace scope,
    global
  global scope, see scope, global
  global-module-fragment, 237, 1560
  glvalue, 82
  goto, 15, 145, 147, 1554
  and handler, 418
  and try block, 418
  initialization and, 148
  grammar, 1545
    regular expression, 1474
  grep, 1447
  group, 427, 1564
  group-part, 427, 1564
  \( H_n \) (Hermite polynomials), 1174
  h-char, 13, 1546
  h-char-sequence, 12, 1546
  h-pp-tokens, 428, 1565
  h-preprocessing-token, 428, 1565
  handler, see exception handling, handler
    handler, 418, 1563
    handler-seq, 418, 1563
  happens after, 73
  happens before, 73
  hard link, 1389
  has-attribute-expression, 428, 1565
  has-include-expression, 428, 1565
  hash
    instantiation restrictions, 681
    hash code, 783
    hash function, 783
    hash tables, see unordered associative containers
  header, 453
    C, 467, 470, 1596
    C library, 456
    C++ library, 453
    importable, 236
    name, 12–13
  header unit, 236
    preprocessing, 432
  header-name, 12, 1545
  header-name-tokens, 428, 1565
  headers
    C library, 1595
    heap with respect to \texttt{comp} and \texttt{proj}, 1071
    Hermite polynomials \( H_n \), 1174
    hex-quad, 10, 1545
    hexadecimal floating literal, see literal,
      hexadecimal floating
    hexadecimal-digit, 16, 1547
    hexadecimal-digit-sequence, 16, 1547
    hexadecimal-escape-sequence, 17, 1548
    hexadecimal-floating-literal, 19, 1548
    hexadecimal-fractional-constant, 19, 1548
    hexadecimal-literal, 15, 1547
    hexadecimal-prefix, 16, 1547
  hiding, see name hiding
  high-order bit, 50
  hosted implementation, 6
  \( I_\nu \) (Bessel functions), 1172
  id
    qualified, 93
  id-expression, 91
id-expression, 91, 1549

identical
  atomic constraints, see atomic constraint, identical
identifier, 13–14, 92, 152
identifier, 13, 1546
identifier label, 142
identifier-list, 428, 1564
identifier-nondigit, 13, 1546
if, 15, 142, 143, 145, 1554
if-group, 427, 1564
if-section, 427, 1564
ill-formed program, see program, ill-formed
immediate function, 157
immediate function context, 139
immediate invocation, 139
immediate subexpression, 69
implementation
  freestanding, 455
  hosted, 455
implementation limits, see limits, implementation
implementation-defined behavior, see behavior, implementation-defined
implementation-dependent, 393, 477, 1343, 1353
implementation-generated, 25
implicit conversion, see conversion, implicit
implicit conversion sequence, see conversion sequence, implicit
implicit object parameter, 308
implicitly-declared default constructor, see constructor, default, 253
implied object argument, 308
  implicit conversion sequences, 308
  non-static member function and, 308
import, 14, 236, 1560
import-seq, 432, 1565
importable header, see header, importable inclusion
  conditional, see preprocessing directive, conditional inclusion
  source file, see preprocessing directives, source-file inclusion
inclusive-or-expression, 131, 1553
incomplete, 127
incompletely-defined object type, see object type, incompletely-defined
increment operator
  overloaded, see overloading, increment operator
independent_bits_engine
  generation algorithm, 1120
  state, 1119
  textual representation, 1120
  transition algorithm, 1119
indeterminate value, 55, see value, indeterminate
indeterminately sequenced, 71
indirect base class, see base class, indirect
indirection, 115
inheritance, 269, 270
  using-declaration and, 214
init-capture, 98, 1550
init-capture pack, 102, 356
init-declarator, 168, 1556
init-declarator-list, 168, 1556
init-statement, 141, 1553
initial suspend point, 202
initialization, 77, 182–198
  aggregate, 187
  array, 187
  array of class objects, 190, 290
  automatic, 148, 149
  automatic object, 183
  base class, 290, 291
  by inherited constructor, 294
  character array, 191
  class member, 184
  class object, see also constructor, 187, 289–295
const, 161, 186
const member, 292
constant, 77
  constructor and, 289
  copy, 185
  default, 183
  default constructor and, 289
  definition and, 152
direct, 185
dynamic, 77
dynamic block-scope, 149
dynamic non-local, 78
explicit, 289
jump past, 148
list-initialization, 193–198
local static, 149
local thread_local, 149
member, 290
  member function call during, 293
  member object, 291
  order of, 78, 271
  order of base class, 292
  order of member, 292
  order of virtual base class, 292
overloaded assignment and, 290
parameter, 107
reference, 174, 191
  reference member, 292
runtime, 77
static and thread, 77
static member, 264
static object, 77, 183
union, 191, 268
vacuous, 53
virtual base class, 256
zero-initialization, 183
initializer
  base class, 198
  member, 198
pack expansion, 294
scope of member, 293
temporary and declarator, 61
initializer, 183, 1557
initializer-clause, 183, 1558
initializer-list, 183, 1558
initializer-list constructor, 194
seed sequence requirement, 1108
<initializer_list>, 502
initializing declaration, 186
injected-class-name, 242
inline, 470
inline function, 159, see function, inline
inline namespace, see namespace, inline
inline namespace set, 209
inline variable, see variable, inline
instantiation
explicit, 391
point of, 382
template implicit, 387
instantiation context, 239
instantiation units, 10
int, 15, 162, 1556
integer literal, see literal, integer
integer representation, 59
integer type, 66
integer-literal, 15, 1546
integer-suffix, 16, 1547
integral constant expression, see expression,
inintegral constant
integral promotion, 88
integral type, 66
implementation-defined sizeof, 65
inter-thread happens before, 73
interface dependency, 237
internal linkage, 48
interval boundaries
piecewise_constant_distribution, 1139
piecewise_linear_distribution, 1141
<inttypes.h>, 1436
invalid pointer value, see value, invalid pointer
invocation
macro, 434
<iomanip>, 1340
<iostream>, 1312
<iterator>, 878, 1604
jν (Bessel functions), 1172
Jessie, 261
jump-statement, 147, 1554
K (complete elliptic integrals), 1171
Kν (Bessel functions), 1173
keyword, 14, 1545
Lν (Laguerre polynomials), 1174
Lν (associated Laguerre polynomials), 1171
label, 148
case, 142, 144
default, 142, 144
scope of, 32, 142
labeled-statement, 142, 1554
Laguerre polynomials
Lν, 1174
Lν, 1171
lambda-capture, 97, 1550
lambda-declarator, 93, 1550
lambda-expression, 93, 1550
lambda-introducer, 93, 162, 1550
language linkage, 219
lattice, see DAG, subobject
layout
bit-field, 265
class object, 248, 271
layout-compatible, 65
class, 248
enumeration, 206
layout-compatible type, 65
left shift
undefined, 128
left shift operator, see operator, left shift
Legendre functions Yνm, 1175
Legendre polynomials
Pν, 1174
Pν, 1171
letter, 451
lexical conventions, see conventions, lexical
LIA-1, 1614
library
C standard, 444, 451, 453, 456, 1591, 1595
C++ standard, 443, 467, 469, 471
library clauses, 7
lifetime, 53
limits
implementation, 4
<limits>, 479
line number, 438
line splicing, 9
linear_congruential_engine
generation algorithm, 1115
modulus, 1115
state, 1115
textual representation, 1115
transition algorithm, 1115
link, 1389
linkage, 24, 47–50

Index
const and, 48
external, 48, 456, 467
implementation-defined object, 222
inline and, 48
internal, 48
module, 48
no, 48, 49
static and, 48
linkage specification, see specification, linkage
linkage-specification, 219, 220, 1559
<list>, 797
list-initialization, 193
literal, 15–23, 90
  base of integer, 16
  binary, 16
  boolean, 22
  char16_t, 17
  char32_t, 17
  character, 17
    ordinary, 17
    UTF-16, 18
    UTF-32, 18
    UTF-8, 17
  wide, 18
  complex, 1103
  constant, 15
  decimal, 16
  decimal floating, 19
double, 19
float, 19
floating, 19
hexadecimal, 16
hexadecimal floating, 19
implementation-defined value of char, 18
integer, 15, 16
long, 16
long double, 19
multicharacter, 17
  implementation-defined value of, 17
narrow-character, 17
octal, 16
operator, 331
  raw, 331
  template, 331
  template numeric, 331
  template string, 331
pointer, 22
string, 19, 20
  char16_t, 20
  char32_t, 20
  narrow, 20
  raw, 11, 20
  undefined change to, 21
  UTF-16, 21
  UTF-32, 21
  UTF-8, 20
  wide, 20, 21
suffix identifier, 331
type of character, 17
type of floating-point, 19
type of integer, 16
unsigned, 16
user-defined, 22
literal, 15, 1546
literal type, 65
literal-operator-id, 331, 1562
living dead
  name of, 465
local class, see class, local
  friend, 287
  member function in, 249
  scope of, 269
local entity, see entity, local
local scope, see block scope
local variable, 32
  destruction of, 147, 148
local_iterator, 785
locale, 1438–1440, 1447
<locale>, 1271, 1272
locale-specific behavior, see behavior,
  locale-specific
<locale.h>, 1309
lock-free execution, 75
logical-and-expression, 131, 1553
logical-or-expression, 131, 1553
lognormal_distribution
  probability density function, 1134
long, 15, 162, 1556
typedef and, 153
long-long-suffix, 16, 1547
long-suffix, 16, 1547
lookup
  argument-dependent, 39
  class member, 42, 46
  elaborated type specifier, 46
  member name, 278
  name, 24, 35–47
  namespace aliases and, 47
  namespace member, 43
  qualified name, 41–46
  template name, 370
  unqualified name, 35
  using-directives and, 47
lookup_classname
  regular expression traits, 1440, 1476
lookup_collatename
  regular expression traits, 1440
low-order bit, 50
lowercase, 451
lparen, 427, 1564
lvalue, 82, 1570
lvalue reference, 173
Lvalue-Callable, 676
macro
  argument substitution, 434
  function-like, 433, 434
  arguments, 434
masking, 470
name, 433
object-like, 433, 434
pragma operator, 441
predefined, 439
replacement, 433–438
replacement list, 433
rescanning and replacement, 436
scope of definition, 436

macro definition, 432
active, 433
point of definition, 432
point of undefinition, 432

macro import, 432–433
point of, 432

main function, 77
implementation-defined linkage of, 77
implementation-defined parameters to, 77
parameters to, 77
return from, 77, 79

make progress
thread, 76

manifestly constant-evaluated, 139

memory location, 50
memory management, see new, delete
memory model, 50–51

max
random number distribution requirement, 1113

mean
normal_distribution, 1133
poisson_distribution, 1130

member
class static, 56
default initializer, 247
eumerator, 207
static, 246, 263
template and static, 354

member access operator
overloaded, 330

member candidate, 311

member data
static, 264

member function, 246
call undefined, 250
class, 249
const, 251
const volatile, 251
constructor and, 252
destructor and, 250
friend, 286
inline, 249
local class, 269
nested class, 289

non-static, 250
overload resolution and, 308
static, 246, 264
this, 251
union, 267
volatile, 251

member of an unknown specialization, 379
member of the current instantiation, see current instantiation, member of the
member pointer to, see pointer to member
member subobject, 51

member-declaration, 246, 1561
member-declarator, 246, 1561
member-declarator-list, 246, 1561

members, 32

<memory>, 597
memory location, 50
memory management, see new, delete
memory model, 50–51

<memory_resource>, 645
merenne_twister_engine
generation algorithm, 1116
state, 1116
textual representation, 1117
transition algorithm, 1116

message
diagnostic, 3, 6

min
random number distribution requirement, 1113
modifiable, 83
modification order, 72
module, 233
exported, 236
global, 233
interface, 234
named, 232

module, 14, 48, 232, 237, 1549, 1560
module implementation unit, 232
module interface unit, 232
module partition, 232
module unit, 232
module unit purview, see purview, module unit

module-declaration, 232, 1560
module-import-declaration, 236, 1560
module-name, 232, 1560
module-name-qualifier, 232, 1560
module-partition, 232, 1560

more constrained, 351
more cv-qualified, 68
more specialized, 363, 409
class template, 363
function template, 409

most derived class, 52
most derived object, 52
bit-field, 52
zero size subobject, 52
move

- class object, see constructor, move, see assignment operator, move

move, 547

multi-pass guarantee, 895, 898

multibyte character, see character, multibyte

multibyte encoding, see encoding, multibyte

multicharacter literal, see literal, multicharacter

multiline, 1447

multiple inheritance, 269, 271

virtual and, 275

multiple threads, see threads, multiple

mutiplicative-expression, 127, 1552

mutable, 15, 153, 1555

mutable iterator, 885

<mutex>, 1507

mutex types, 1508

\( n_n \) (spherical Neumann functions), 1175

\( N_\nu \) (Neumann functions), 1173

name, 13, 24, 91

- address of cv-qualified, 115

- dependent, 376

- elaborated

enum, 162

exported, 235

global, 33

- length of, 13

- macro, see macro, name

- point of declaration, 30

- predefined macro, see macro, predefined qualified, 41

- reserved, 465

- same, 24

- scope of, 30

- unqualified, 35

- zombie, 465

name class, see class name

name hiding, 31, 35, 92, 148

- class definition, 244

- function, 306

- overloading versus, 306

- user-defined conversion and, 261

- using-declaration and, 217

name space

- label, 142

named module, see module, named

named-name-space-definition, 208, 1559

namespace, 453, 1596

- alias, 211

- associated, 39

- definition, 208

- enclosing, 208

- extend, 208

- global, 14, 33

- inline, 208

- member definition, 210

- unnamed, 209

namespace, 15, 208, 209, 211, 1559

namespace scope

- global, 33

namespace-alias, 211, 1559

namespace-alias-definition, 211, 1559

namespace-body, 208, 1559

namespace-definition, 208, 1559

namespace-name, 208, 1559

namespaces, 207–214

NaN, 1170

narrow character type, 66

narrowing conversion, 197

native encoding, 1399

native pathname format, 1394

NDEBUG, 456

needed

- exception specification, 424

- needed for constant evaluation, 140

negative_binomial_distribution

- discrete probability function, 1129

nested class, see class, nested

- local class, 269

- scope of, 265

- nested within, 52

nested-name-specifier, 92, 1550

nested-name-space-definition, 208, 1559

nested-requirement, 105, 1551

Neumann functions

- \( N_\nu \), 1173

- \( n_n \), 1175

<new>, 490

new, 15, 57, 119, 120, 381, 1552

- array of class objects and, 122

- constructor and, 122

- default constructor and, 122

- exception and, 122

- initialization and, 122

operator

- replaceable, 467, 468

- scoping and, 119

- storage allocation, 119

- type of, 302

- unspecified constructor and, 122

- unspecified order of evaluation, 122

new-declarator, 119, 1552

new-expression, 119, 1552

- placement, 121

new-extended alignment, 60

ew-initializer, 119, 1552

ew-line, 428, 1564

new-placement, 119, 1552

new-type-id, 119, 1552

new_handler, 58

no linkage, 48

node handle, 770

nodeclspec-function-declaration, 151, 1555

noexcept, 15, 104, 125, 381, 382, 1551, 1552

noexcept-expression, 125, 1552

noexcept-specifier, 423, 1563
non-initialization odr-use, see odr-use, non-initialization
non-member candidate, 311
non-static data member, 246
non-static member, 246
non-static member function, 246
non-throwing exception specification, 423
non-virtual base class, see base class, non-virtual nondigit, 13, 1546
nonzero-digit, 16, 1547
noptr-abstract-declarator, 170, 1557
noptr-abstract-pack-declarator, 170, 1557
noptr-declarator, 169, 1556
noptr-new-declarator, 119, 1552
normal distributions, 1133–1138
normal form
constraint, 350
path, 1398
normal_distribution
mean, 1133
probability density function, 1133
standard deviation, 1133
normalization
constraint, see constraint, normalization
path, see path, normalization
normative references, see references, normative
not, 15
not_eq, 15
notation
syntax, 7–8
NTBS, 1602
NTBS, 452
NTBS
empty, 452
length, 452
static, 452
value, 452
NTCTS, 445
NTMBS, 452
static, 452
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, 142
null wide character, see wide-character, null
nullptr, 15
number
hex, 18
octal, 18
preprocessing, 13
subnormal, 480, 481, 483, 484
<numeric>, 1080
numeric type, see type, numeric
numeric_limits, 479
specializations for arithmetic types, 66
object, see also object model, 24, 51
byte copying and, 63–64
callable, 661
complete, 51
const, 68
const volatile, 68
definition, 26
destructor and placement of, 260
destructor static, 79
exception, see exception handling, exception
object
linkage specification, 222
local static, 56
nested within, 52
nonzero size, 52
providing storage for, 51
unnamed, 252
volatile, 68
zero size, 52
object class, see class object
object expression, 108, 126
object lifetime, 53–55
object model, 51–52
object pointer type, 67
object temporary, see temporary
object type, 64
incompletely-defined, 64
object-like macro, see macro, object-like
observable behavior, see behavior, observable
octal-digit, 16, 1547
octal-escape-sequence, 17, 1548
octal-literal, 15, 1547
odr-usable, 27
odr-use, 27
non-initialization, 78
one-definition rule, 26–30
opaque-enum-declaration, 205, 1558
operating system dependent, 1390
operation
atomic, 72–77
operator, 14, 329
**=, 134
+=, 116, 134
-=, 134
/=, 134
<<=, 134
>>=, 134
%=, 134
&=, 134
|=, 134
^=, 134
addition, 127
additive, 127
address-of, 115
assignment, 134, 452
bitwise, 131
bitwise AND, 131
bitwise exclusive OR, 131
bitwise inclusive OR, 131
cast, 115, 170
class member access, 108
comma, 135
conditional expression, 132
copy assignment, see assignment operator,
copy
decrement, 109, 115, 116
division, 127
equality, 130
defaulted, 301
function call, 106, 328
greater than, 129
greater than or equal to, 129
implementation, 328
increment, 109, 115, 116
indirection, 115
inequality, 130
defaulted, 301
left shift, 128
less than, 129
less than or equal to, 129
logical AND, 131
logical negation, 115, 116
logical OR, 131
move assignment, see assignment operator,
multiply
multiplication, 127
multiplicative, 127
ones’ complement, 115, 116
overloaded, 81, 328
pointer to member, 126
pragma, see macro, pragma operator
precedence of, 81
relational, 129
defaulted, 302
remainder, 127
right shift, 128
scope resolution, 41, 92, 120, 249, 270, 276
side effects and comma, 135
side effects and logical AND, 131
side effects and logical OR, 132
sizeof, 115, 118
spaceship, 128
subscripting, 106, 328
subtraction, 127
three-way comparison, 128
defaulted, 301
unary, 115
unary minus, 115, 116
unary plus, 115, 116
operator, 328, 1562
operator, 15, 262, 310, 328, 331, 1561, 1562
operator delete
destroying, 58
operator delete, see also delete, 120, 124, 302
operator new, see also new, 120
operator overloading, see overloading, operator
operator use
scope resolution, 264
operator!=
Index

binary operator, 329
built-in operators and, 332
candidate functions, 307–315
declaration matching, 306
declarations, 304
element of, 304
function call operator, 330
function versus pointer, 305
member access operator, 330
operator, 328–332
prohibited, 304
resolution, 307–326
best viable function, 316–329
better viable function, 316
contexts, 307
function call syntax, 309–310
function template, 416
implicit conversions and, 318–326
initialization, 313, 314
operators, 310
scoping ambiguity, 278
template, 367
template name, 370
viable functions, 315–329
subscripting operator, 330
unary operator, 329
user-defined literal, 331
using directive and, 213
using-declaration and, 218
overloads
floating-point, 1103
override, 14, 246, 1561
overrider
final, 273
own, 618
P ell (Legendre polynomials), 1174
Pm l (associated Legendre polynomials), 1171
pack, 357
unexpanded, 357
pack expansion, 357
pattern, 357
padding bits, 64
tuple interface to, 548
parallel algorithm, 988
parallel forward progress guarantees, 76
param random number distribution requirement, 1113
seed sequence requirement, 1109
param_type random number distribution requirement, 1112
parameter, 4
catch clause, 4
function, 4
function-like macro, 4
macro, 434
reference, 173
scope of, 32
template, 4, 25
void, 177
parameter declaration, 25
parameter list variable, 108, 177
parameter mapping, 348
parameter-declaration, 177, 1557
parameter-declaration-clause, 176, 1557
parameter-declaration-list, 177, 1557
parameter-type-list, 177
parameterized type, see template
parameters-and-qualifiers, 169, 1557
parent directory, 1389
path, 1394
normalization, 1398
path equality, 1408
pathname, 1394
pathname, 1397
pathname resolution, 1394
pattern, see pack expansion, pattern
perfect forwarding call wrapper, 662
period, 451
phases of translation, see translation, phases
Π (complete elliptic integrals), 1172
Π (incomplete elliptic integrals), 1174
piecewise construction, 550
piecewise_constant_distribution
interval boundaries, 1139
probability density function, 1139
weights, 1139
piecewise_linear_distribution
interval boundaries, 1141
probability density function, 1140
weights at boundaries, 1141
placeholder type deduction, 167
placeholder-type-specifier, 165, 1556
placement new-expression, see new-expression, placement
plain lock-free atomic operation, 516
pm-expression, 126, 1552
POD, 1604
point, 67
point of declaration, 30
point of macro import, see macro import, point of
pointer, see also void*
composite pointer type, 84
integer representation of safely-derived, 59
safely-derived, 59
to traceable object, 59, 472
zero, see value, null pointer
pointer literal, see literal, pointer
pointer past the end of, 67
pointer to, 67
pointer to member, 67, 126, 174
pointer-interconvertible, 68
pointer-literal, 22, 1549
Poisson distributions, 1130–1133
poisson_distribution
  discrete probability function, 1130
  mean, 1130
polymorphic class, see class, polymorphic
pool resource classes, 650
pools, 650
population, 1053
POSIX, 2
  extended regular expressions, 1447
  regular expressions, 1447
postcondition, 225
postfix ++ and --
  overloading, 330
postfix ++, 109
postfix --, 109
postfix-expression, 106, 1551
potential results, 26
potential scope, 30
potentially concurrent, 74
potentially constant evaluated, 139
potentially evaluated, 26
potentially-overlapping subobject, 52
potentially-throwing
  exception specification, 422
expression, 423
pp-balanced-token, 432, 1565
pp-balanced-token-seq, 432, 1565
pp-global-module-fragment, 432, 1565
pp-import, 432, 1565
pp-import-suffix, 432, 1565
pp-import-suffix-token, 432, 1565
pp-ldelimiter, 432, 1565
pp-number, 13, 1546
pp-rdelimiter, 432, 1565
pp-tokens, 428, 1564
precedence of operator, see operator, precedence of
precondition, 225
preferred-separator, 1397
prefix
  L, 18, 21
  R, 20
  U, 18, 21
  u, 18, 21
  u8, 17, 20
prefix ++ and --
  overloading, 330
prefix ++, 116
prefix --, 116
preprocessing, 428
preprocessing directive, 427–442
  conditional inclusion, 428
  error, 439
  header inclusion, 431
  line control, 438
  macro replacement, see macro, replacement
  null, 439
  pragma, 439
  source-file inclusion, 431
preprocessing-file, 427, 1564
preprocessing-op-or-punc, 14, 1546
preprocessing-token, 11, 1545
primary class template, see template, primary
primary equivalence class, 1438
primary module interface unit, 232
primary-expression, 90, 1549
private, 15, 48, 270, see access control, private, 1549, 1561
private-module-fragment, 48, 1549
probability density function
  cauchy_distribution, 1136
  chi_squared_distribution, 1135
  exponential_distribution, 1130
  extreme_value_distribution, 1133
  fisher_f_distribution, 1136
  gamma_distribution, 1131
  lognormal_distribution, 1134
  normal_distribution, 1133
  piecewise_constant_distribution, 1139
  piecewise_linear_distribution, 1140
  student_t_distribution, 1137
  uniform_real_distribution, 1126
  weibull_distribution, 1132
program, 47
  ill-formed, 4
  start, 77–79
  termination, 79–80
  well-formed, 5, 7
program execution, 6–72
  abstract machine, 6
  as-if rule, see as-if rule
program semantics
  affected by the existence of a variable or function definition, 389
projection, 446
promise object, 202
promoted arithmetic type, 332
promoted integral type, 332
promotion
  bool to int, 88
  default argument promotion, 108
  floating-point, 88
  integral, 87
protected, 15, 270, see access control, protected, 1561
protection, see access control, 471
prototype parameter
  concept, 370
  provides storage, 51
prvalue, 82
pseudo-destructor, 93
ptr-abstract-declarator, 170, 1557
ptr-declarator, 169, 1556
ptr-operator, 170, 1557
ptrdiff_t, 127
  implementation-defined type of, 127
public, 15, 270, see access control, public, 1561
punctorator, 14
Index
pure-specifier, 246, 1561
purview
  global module, 233
  module unit, 233
  named module, 233
q-char, 13, 1546
q-char-sequence, 13, 1546
qualification
  explicit, 41
qualified-id, 92, 1550
qualified-namespace-specifier, 211, 1559
<queue>, 864
r-char, 20, 1548
r-char-sequence, 20, 1548
<random>, 1105
random number distribution
  bernoulli_distribution, 1127
  binomial_distribution, 1127
  cauchy_distribution, 1136
  chi_squared_distribution, 1135
  discrete_distribution, 1138
  exponential_distribution, 1130
  extreme_value_distribution, 1133
  fisher_f_distribution, 1136
  gamma_distribution, 1131
  geometric_distribution, 1128
  lognormal_distribution, 1134
  negative_binomial_distribution, 1129
  normal_distribution, 1133
  piecewise_constant_distribution, 1139
  piecewise_linear_distribution, 1140
  poisson_distribution, 1130
  requirements, 1112–1114
  student_t_distribution, 1137
  uniform_int_distribution, 1125
  uniform_real_distribution, 1126
  weibull_distribution, 1132
random number distributions
  Bernoulli, 1127–1130
  normal, 1133–1138
  Poisson, 1130–1133
  sampling, 1138–1142
  uniform, 1125–1127
random number engine
  linear_congruential_engine, 1115
  mersenne_twister_engine, 1116
  requirements, 1109–1111
  subtract_with_carry_engine, 1117
  with predefined parameters, 1121–1122
random number engine adaptor
  discard_block_engine, 1118
  independent_bits_engine, 1119
  shuffle_order_engine, 1120
  with predefined parameters, 1121–1122
random number generation, 1105–1142
  distributions, 1125–1142
  engines, 1114–1121
predefined engines and adaptors, 1121–1122
  requirements, 1107–1114
  synopsis, 1105–1107
  utilities, 1123–1125
random number generator, see uniform random bit generator
random_device
  implementation leeway, 1122
<ratio>, 706
raw string literal, 20
raw-string, 20, 1548
reachable
  declaration, 240
  necessarily
    declaration, 240
    translation unit, 240
    translation unit, 240
ready, 1461, 1534
redefinition
typedef, 156
ref-qualifier, 170, 1557
reference, 67
  assignment to, 134
  call by, 107
  forwarding, 405
  lvalue, 67
  null, 174
  rvalue, 67
sizeof, 118
reference collapsing, 174
reference lifetime, 53
reference-compatible, 192
reference-related, 192
references
  normative, 2
<regex>, 1440
regex_iterator
  end-of-sequence, 1470
regex_token_iterator
  end-of-sequence, 1472
regex_traits
  end-of-sequence, 1472
region
  declarations, 1449
  specializations, 1449
register
  declaration, 24, 30
  intervening, 30
register storage class, 1583
regular expression, 1438–1476
  grammar, 1474
  matched, 1438
  requirements, 1439
regular expression traits, 1474
  char_class_type, 1439
  isctype, 1440
  lookup_classname, 1440, 1476
  lookup_collatename, 1440
  requirements, 1439, 1449
  transform, 1439, 1476
  transform_primary, 1440, 1476
translate, 1439, 1476
translate_nocase, 1439, 1476
reinterpret_cast, 15, see cast, reinterpret, 106, 381, 382, 1551
relational-expression, 129, 1552
relative path, 1394
relative-path, 1397
relaxed pointer safety, 59
release sequence, 72
remainder operator, see operator, remainder
remote time zone database, 1252
replacement
macro, see macro, replacement
replacement-list, 428, 1564
representation
object, 64
value, 64
represents the address, 67
requirement, 103
compound, 104
nested, 105
simple, 104
type, 104
requirement, 103, 1551
requirement-body, 103, 1551
requirement-parameter-list, 103, 1551
requirement-seq, 103, 1551
requirements, 447
container, 700, 783, 799, 800, 1460
not required for unordered associated containers, 783
Cpp17Allocator, 459
Cpp17CopyAssignable, 456
Cpp17CopyConstructible, 456
Cpp17DefaultConstructible, 456
Cpp17Destructible, 456
Cpp17EqualityComparable, 456
Cpp17Hash, 459
Cpp17LessThanComparable, 456
Cpp17MoveAssignable, 456
Cpp17MoveConstructible, 456
Cpp17NullablePointer, 459
iterator, 884
numeric type, 1094
random number distribution, 1112–1114
random number engine, 1109–1111
regular expression traits, 1439, 1449
seed sequence, 1108–1109
sequence, 1460
uniform random bit generator, 1109
unordered associative container, 783
requires, 15, 103, 105, 335, 1551, 1562
requires-clause, 335, 1562
trailing, 169
requires-expression, 103, 1551
rescanning and replacement, see macro, rescanning and replacement
reserved identifier, 14
reset, 618
reset
random number distribution requirement, 1113
resolution, see overloading, resolution
restriction, 469–471
address of bit-field, 265
anonymous union, 268
bit-field, 265
constructor, 253
destructor, 258
extern, 154
local class, 269
operator overloading, 328
overloading, 328
pointer to bit-field, 265
reference, 174
static, 154
static member local class, 269
union, 267
result
glvalue, 83
prvalue, 83
result object, 83
result_type
entity characterization based on, 1105
random number distribution requirement, 1112
seed sequence requirement, 1108
rethrow, see exception handling, rethrow
return, 15, 147, 1554
and handler, 418
and try block, 418
constructor and, 147
reference and, 191
return statement, see return
return type, 178
covariant, 274
overloading and, 304
return-type-requirement, 104, 1551
rewritten candidate, 311
right shift operator, see operator, right shift
root-directory, 1397
root-name, 1397
rounding, 88
rvalue, 82
lvalue conversion to, see conversion, 1570
lvalue-to-rvalue, 1570
rvalue reference, 173
s-char, 20, 1548
s-char-sequence, 20, 1548
safely-derived pointer, 59
integer representation, 59
sample, 1053
sampling distributions, 1138–1142
scalar type, 64
scope, 1, 24, 30–35, 152
anonymous union at namespace, 268
block, 32
class, 33
declarations and, 30–32
destructor and exit from, 147
enumeration, 34
exception declaration, 32
function, 32
function parameter, 32
function prototype, see scope, function parameter
global, 33
iteration-statement, 144
macro definition, see macro, scope of definition
name lookup and, 35–47
namespace, 32
overloading and, 306
potential, 30
selection-statement, 142
template parameter, 34
scope name hiding and, 35
scope resolution operator, see operator, scope resolution
scoped enumeration, see enumeration, scoped
<scoped_allocator>, 654
seed
  random number engine requirement, 1110
seed sequence, 1108
requirements, 1108–1109
selection-statement, 142, 1554
semantics
  class member, 108
separate compilation, see compilation, separate
separate translation, see compilation, separate
sequence constructor
  seed sequence requirement, 1108
sequenced after, 71
sequenced before, 71
sequencing operator, see operator, comma
<set>, 824
<setjmp.h>, 516
setlocale, 451
shared lock, 1512
shared mutex types, 1512
shared state, see future, shared state
shared timed mutex type, 1513
<shared_mutex>, 1507
shift operator
  left, see operator, left shift
  right, see operator, right shift
shift-expression, 128, 1552
short, 15, 162, 1556
typedef and, 153
shuffle_order_engine
  generation algorithm, 1120
  state, 1120
textual representation, 1121
transition algorithm, 1120
side effect
  visible, 74
side effects, 6, 61, 71–75, 131, 132, 142, 292, 298, 437, 471
  visible, 74
sign, 19, 1548
signal, 71
signal-safe
  _Exit, 488
  abort, 488
evaluation, see evaluation, signal-safe
  forward, 547
initializer_list functions, 502
memcpy, 756
memmove, 756
move, 547
move_if_noexcept, 547
numeric_limits members, 481
quick_exit, 489
signal, 517
type traits, 681
<signal.h>, 516
signature, 4, 5
signed, 15, 162, 1556
typedef and, 153
signed integer representation
  ones’ complement, 116
  two’s complement, 65, 206, 707, 1486, 1494
signed integer type, 65
significand, 19
similar types, 87
simple call wrapper, 661
simple-capture, 98, 1550
simple-declaration, 151, 1555
simple-escape-sequence, 17, 1547
simple-requirement, 104, 1551
simple-template-id, 340, 1563
simple-type-specifier, 162, 1556
simply happens before, 73
size
  seed sequence requirement, 1108
size_t, 118
sizeof, 15, 115, 381, 382, 1551
smart pointers, 627–642
source file, 9, 456, 467
source file character, see character, source file space
  white, 11
<span>, 872
special member function, see constructor, see assignment operator, see destructor
specialization
  class template, 341
  class template partial, 361
program-defined, 446
template, 386
template explicit, 394
specification
  linkage, 219–222
  extern, 219
implementation-defined, 220
nesting, 220
template argument, 399
specifications
  C standard library exception, 471
  C++, 472
specifier, 152–168
  consteval, 157
  constexpr, 157
  constructor, 157, 158
  function, 157
cv-qualifier, 161
declaration, 152
  explicit, 155
  friend, 157, 471
  function, 155
  inline, 159
  static, 153
  storage class, 153
type, see type specifier
typedef, 155
virtual, 155
specifier access, see access specifier
spherical harmonics $Y^m_\ell$, 1175
<sstream>, 1362
stable algorithm, 447, 470
<stack>, 865
stack unwinding, 420
standard
  structure of, 7
standard deviation
  normal_distribution, 1133
standard integer type, 66
standard signed integer type, 65
standard unsigned integer type, 65
standard-layout class, see class, standard-layout
standard-layout struct, see struct, standard-layout
standard-layout types, 65
standard-layout union, see union, standard-layout
start
  program, 78
startup
  program, 456, 468
state, 587
  discard_block_engine, 1118
  independent_bits_engine, 1119
  linear_congruential_engine, 1115
  mersenne_twister_engine, 1116
  shuffle_order_engine, 1120
  subtract_with_carry_engine, 1117
state entity, 661
statement, 141–150
  continue in for, 146
  break, 147
  compound, 142
  continue, 147
  declaration, 148
  declaration in for, 146
  declaration in if, 141
  declaration in switch, 141, 144
declaration in while, 145
do, 144, 145
empty, 142
expression, 142
fallthrough, 229
for, 144, 145
goto, 142, 147, 148
if, 142, 143
iteration, 144–147
jump, 147
labeled, 142
null, 142
range based for, 146
selection, 142–144
switch, 142, 144, 147
while, 144, 145
statement, 141, 1553
statement-seq, 142, 1554
static, 15, 153, 1555
destruction of local, 149
linkage of, 48, 154
overloading and, 304
static data member, 246
static initialization, 77
static member, 246
static member function, 246
static storage duration, 56
static type, see type, static
static_assert, 152
static_assert, 15, 151, 1555
  not macro, 533
static_assert-declaration, 151, 1555
static_cast, 15, see cast, static, 106, 381, 382,
  1551
<stdark.h>, 1592, 1595
<stdark.h>, 516
<stdark.h>
  absence thereof, 453, 1591
<stdbool.h>, 1592, 1595
<stdbexcep.h>, 18, 21
<stdbexcep.h>, 530
<stdark.h>, 1435
<stdark.h>, 1596
<stdark.h>
  absence thereof, 453, 1591
storage class, 24
storage duration, 56–59
  automatic, 56, 57
  class member, 59
dynamic, 56–59, 119
local object, 57
static, 56
  thread, 56, 57
storage management, see new, delete
storage-class-specifier, 153, 1555
stream
  arbitrary-positional, 444
  repositionable, 446
<streambuf>, 1331
templated entity, 336
temporary, 60
corstructor for, 61
destruction of, 61
destructor for, 61
elimination of, 60, 298
implementation-defined generation of, 60
order of destruction of, 61
terminate, 425, 426
called, 134, 420, 423, 425
termination program, 77, 80
terminology
pointer, 67
text-line, 427, 1564
textual representation
discard_block_engine, 1119
independent_bits_engine, 1120
shuffle_order_engine, 1121
subtract_with_carry_engine, 1117
translate
regular expression traits, 1439, 1476
translate_nocase
regular expression traits, 1439, 1476
transition
phases, 9–10
separate, see compilation, separate
translation unit, 9, 48
name and, 24
translation-unit, 48, 1549
trigraph sequence, 1583
trivial class, see class, trivial
trivial types, 65
trivially copyable class, see class, trivially copyable
trivially copyable types, 65
true, 15
truncation, 88
try, 15, 418
try block, see exception handling, try block
try-block, 418, 1563
<tuple>, 553
tuple
and pair, 548
type, 24, 63–69
allocated, 119
arithmetic, 66
promoted, 332
array, 67
bitmask, 450, 451
Boolean, 66
callable, 661
class, 66
cchar, 66
cchar16_t, 18, 21, 66, 69
cchar32_t, 18, 21, 66, 69
cchar8_t, 66
ccharacter, 66
ccharacter container, 444
class and, 242
compound, 67
cconst, 160
cv-combined, 83
cv-unqualified, 68
destination, 185
double, 66
dynamic, 3
enumerated, 67, 450
each example of incomplete, 64
extended integer, 66
extended signed integer, 65
extended unsigned integer, 65
floating-point, 66
function, 67, 176, 177
fundamental, 67
implementation-defined sizeof, 65
incomplete, 26, 28, 31, 64, 86, 106–109, 115, 118, 119, 123, 134, 270

Index
incompletely-defined object, 64
double, 162
int, 65
evaluated, 450
long, 65
long double, 66
long long, 65
narrow character, 66
numeric, 1094
ordinary character, 66
over-aligned, 60
pointer, 67
polymorphic, 272
program-defined, 446
referenceable, 446
short, 65
signed char, 65, 66
signed integer, 65
similar, see similar types
standard integer, 66
standard signed integer, 65
standard unsigned integer, 65
static, 5
trivially copyable, 63
underlying
  char16_t, 66, 87
  char32_t, 66, 87
  char8_t, 66
enumeration, 87, 206
fixed, 206
wchar_t, 66, 87
unsigned, 65
unsigned char, 65, 66
unsigned int, 65
unsigned integer, 65
unsigned long, 65
unsigned long long, 65
unsigned short, 65
void, 66
volatile, 160
wchar_t, 18, 21, 66, 69
type checking
  argument, 107
  type concept, see concept, type
type conversion, explicit, see casting
type generator, see template
type name, 170
  nested, 266
  scope of, 266
type pun, 114
type specifier
  auto, 165
  bool, 162
  char, 162
  char16_t, 162
  char32_t, 162
  char8_t, 162
  const, 161
dcltype, 164
dcltype(auto), 165
double, 162
evaluated, 46, 162
evaluated, 450
float, 162
int, 162
long, 162
short, 162
signed, 162
simple, 161
unsigned, 162
void, 162
volatile, 161
wchar_t, 162
type-constraint, 335, 1562
type-id, 170, 1557
type-id-only context, 371
type-name, 162, 1556
type-parameter, 337, 1563
type-parameter-key, 337, 1563
type-requirement, 104, 1551
type-specifier, 160, 1555
type-specifier-seq, 160, 1555
type_info, 111
<typename_traits>, 682, 1604
typedef
  function, 178
typedef, 15, 153, 1555
  overloading and, 305
typedef-name, 155, 1555
typeid, 15, 106, 111, 381, 382, 1551
  construction and, 297
  destruction and, 297
<typenameindex>, 708
<typenameinfo>, 496
typename, 15, 104, 162, 214, 337, 371, 1551, 1559, 1563
typename-specifier, 371, 1563
types
  implementation-defined, 450
<typename.h>, 758
<uchar.h>, 758
 ud-suffix, 22, 1549
unary fold, 102
unary left fold, 102
unary operator
  interpretation of, 329
  overloaded, 329
unary right fold, 102
unary-expression, 115, 1551
unary-operator, 115, 1552
unblock, 5
undefined, 446, 465, 467, 469, 1153, 1157, 1160, 1324
undefined behavior, see behavior, undefined
underlying type, 66, see type, underlying
unevaluated operand, 84
Unicode required set, 441
uniform distributions, 1125–1127
uniform random bit generator requirements, 1109
uniform_int_distribution discrete probability function, 1125
uniform_real_distribution probability density function, 1126
union, 243
standard-layout, 244
union, 15, 67, 242, 267, 268, 1561
anonymous, 268
global anonymous, 268
union-like class, 268
unique pointer, 618
unit
translation, 456, 467
universal character name, 9
universal-character-name, 10, 1545
Unix time, 1205
unnamed bit-field, 265
unnamed-namespace-definition, 208, 1559
unordered associative containers, 783
complexity, 783
equality function, 783
equivalent keys, 783, 850, 859
exception safety, 795
hash function, 783
iterator invalidation, 795
iterators, 795
lack of comparison functions, 783
requirements, 783, 795
unique keys, 783, 843, 855
unordered_map, 841
unordered_map
element access, 848
unique keys, 843
unordered_multimap
equivalent keys, 850
unordered_multiset
equivalent keys, 850
unordered_set, 842
unordered_set
unique keys, 855
unqualified-id, 92, 1550
unscoped enumeration, see enumeration, unscoped
unsequenced, 71
unsigned, 15, 162, 1556
typedef and, 153
unsigned integer type, 65
unsigned-suffix, 16, 1547
unspecified, 491, 492, 497, 1057, 1366, 1598–1600
unspecified behavior, see behavior, unspecified
unwinding
stack, 420
uppercase, 14, 451
upstream, 653
upstream allocator, 650
usable in constant expressions, 136
user-defined conversion sequence, see conversion
sequence, user-defined
user-defined literal, see literal, user-defined
overloaded, 331
user-defined-character-literal, 22, 1549
user-defined-floating-literal, 22, 1549
user-defined-integer-literal, 22, 1549
user-defined-literal, 22, 1549
user-defined-string-literal, 22, 1549
user-provided, 200
uses-allocator construction, 607
using, 15, 151, 209, 211, 214, 222, 1555, 1559
using-declaration, 214–219
using-declaration, 214, 1559
using-declarator, 214, 1559
using-declarator-list, 214, 1559
using-directive, 211–214
using-directive, 211, 1559
usual arithmetic conversions, see conversion, usual
arithmetic
usual deallocation function, 58
UTF-16 character literal, 18
UTF-16 string literal, 21
UTF-32 character literal, 18
UTF-32 string literal, 21
UTF-8 character literal, 17
UTF-8 string literal, 20
utility, 544, 1596
va-opt-replacement, 434, 1565
vacuous initialization, 53
variant, 1142
valid but unspecified state, 447
value, 64
call by, 107
denormalized, see number, subnormal
indeterminate, 55
invalid pointer, 67
null member pointer, 89
null pointer, 67, 88
undefined unrepresentable integral, 88
value category, 82
value computation, 61, 71, 74, 75, 109, 122, 131,
132, 134, 135
value-initialization, 184
variable, 24
function-local predefined, 198
indeterminate uninitialized, 183
inline, 159
needed for constant evaluation, 140
program semantics affected by the existence
of a variable definition, 389
variable arguments, 434
variable template
definition of, 335
variant, 575
variant member, 268
vector, 798
vectorization-unsafe, 988
violation continuation mode, 228
violation handler, 228
Index
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.

<table>
<thead>
<tr>
<th>Production</th>
<th>Page Numbers</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract-declarator</td>
<td>170, 170, 177, 179, 180, 1557</td>
<td></td>
</tr>
<tr>
<td>abstract-pack-declarator</td>
<td>170, 1557</td>
<td></td>
</tr>
<tr>
<td>access-specifier</td>
<td>270, 270, 282, 283, 1561</td>
<td></td>
</tr>
<tr>
<td>additive-expression</td>
<td>127, 1552</td>
<td></td>
</tr>
<tr>
<td>alias-declaration</td>
<td>25, 50, 151, 155, 156, 177, 235, 238, 246, 335, 352, 369, 1555</td>
<td></td>
</tr>
<tr>
<td>alignment-specifier</td>
<td>222, 223, 224, 357, 1559</td>
<td></td>
</tr>
<tr>
<td>and-expression</td>
<td>131, 1553</td>
<td></td>
</tr>
<tr>
<td>assignment-expression</td>
<td>134, 152, 165, 167, 180, 187, 190, 204, 264, 289, 313, 1553</td>
<td></td>
</tr>
<tr>
<td>assignment-operator</td>
<td>134, 1553</td>
<td></td>
</tr>
<tr>
<td>attribute</td>
<td>222, 223, 227, 1559</td>
<td></td>
</tr>
<tr>
<td>attribute-argument-clause</td>
<td>222, 222, 223, 224, 229–231, 1560</td>
<td></td>
</tr>
<tr>
<td>attribute-declaration</td>
<td>25, 151, 152, 1555</td>
<td></td>
</tr>
<tr>
<td>attribute-list</td>
<td>222, 223, 224, 229–31, 357, 1559</td>
<td></td>
</tr>
<tr>
<td>attribute-name</td>
<td>222, 223, 223, 1560</td>
<td></td>
</tr>
<tr>
<td>attribute-scoped-token</td>
<td>222, 222, 223, 1560</td>
<td></td>
</tr>
<tr>
<td>attribute-specifier</td>
<td>222, 222, 223, 1559</td>
<td></td>
</tr>
<tr>
<td>attribute-token</td>
<td>14, 222, 222, 223, 224, 229–231, 429, 467, 1560</td>
<td></td>
</tr>
<tr>
<td>attribute-using-prefix</td>
<td>222, 222, 1559</td>
<td></td>
</tr>
<tr>
<td>await-expression</td>
<td>117, 117, 137, 201, 209, 515, 1552</td>
<td></td>
</tr>
<tr>
<td>balanced-token</td>
<td>222, 1560</td>
<td></td>
</tr>
<tr>
<td>balanced-token-seq</td>
<td>222, 223, 1560</td>
<td></td>
</tr>
<tr>
<td>base-clause</td>
<td>242, 269, 1561</td>
<td></td>
</tr>
<tr>
<td>base-specifier</td>
<td>36, 42, 269, 270, 278, 281, 285, 357, 1561</td>
<td></td>
</tr>
<tr>
<td>base-specifier-list</td>
<td>258, 269, 270, 281, 292, 294, 301, 357, 1561</td>
<td></td>
</tr>
<tr>
<td>binary-digit</td>
<td>16, 1547</td>
<td></td>
</tr>
<tr>
<td>binary-exponent-part</td>
<td>19, 1548</td>
<td></td>
</tr>
<tr>
<td>binary-literal</td>
<td>15, 1547</td>
<td></td>
</tr>
<tr>
<td>block-declaration</td>
<td>151, 1555</td>
<td></td>
</tr>
<tr>
<td>boolean-literal</td>
<td>22, 1549</td>
<td></td>
</tr>
<tr>
<td>brace-or-equal-initializer</td>
<td>69, 175, 183, 183, 185, 187, 247, 253, 264, 380, 1558</td>
<td></td>
</tr>
<tr>
<td>braced-init-list</td>
<td>61, 69, 106, 108, 120, 135, 139, 148, 167, 183, 185, 189, 193, 194, 197, 200, 291, 293, 376, 381, 397, 1558, 1567, 1583</td>
<td></td>
</tr>
<tr>
<td>c-char</td>
<td>17, 17, 18, 21, 1547</td>
<td></td>
</tr>
<tr>
<td>c-char-sequence</td>
<td>10, 17, 1547</td>
<td></td>
</tr>
<tr>
<td>capture</td>
<td>98, 357, 1550</td>
<td></td>
</tr>
<tr>
<td>capture-default</td>
<td>27, 97, 98–100, 1550, 1588, 1594</td>
<td></td>
</tr>
<tr>
<td>capture-list</td>
<td>98, 357, 1550</td>
<td></td>
</tr>
<tr>
<td>cast-expression</td>
<td>102, 117, 123, 125, 125, 126, 139, 303, 357, 358, 382, 1552</td>
<td></td>
</tr>
<tr>
<td>character-literal</td>
<td>17, 1547</td>
<td></td>
</tr>
<tr>
<td>class-head</td>
<td>31, 223, 242, 1560</td>
<td></td>
</tr>
<tr>
<td>class-head-name</td>
<td>210, 242, 341, 1560</td>
<td></td>
</tr>
<tr>
<td>class-key</td>
<td>46, 152, 163, 210, 242, 243–245, 283, 353, 1561</td>
<td></td>
</tr>
<tr>
<td>class-name</td>
<td>8, 31, 35, 46, 47, 152, 156, 162, 163, 242, 245, 258, 341, 1560</td>
<td></td>
</tr>
<tr>
<td>class-or-dectype</td>
<td>242, 270, 270, 290, 291, 341, 371, 1561</td>
<td></td>
</tr>
<tr>
<td>class-specifier</td>
<td>31, 152, 160, 242, 247, 252, 1560</td>
<td></td>
</tr>
<tr>
<td>class-virt-specifier</td>
<td>242, 491, 1561</td>
<td></td>
</tr>
<tr>
<td>compare-expression</td>
<td>128, 1552</td>
<td></td>
</tr>
<tr>
<td>compound-requirement</td>
<td>104, 105, 1551</td>
<td></td>
</tr>
<tr>
<td>compound-statement</td>
<td>70, 77, 97, 98, 101, 102, 117, 141, 142, 145, 158, 253, 292, 390, 418, 419, 421, 1554</td>
<td></td>
</tr>
<tr>
<td>concept-definition</td>
<td>335, 370, 1562</td>
<td></td>
</tr>
<tr>
<td>concept-name</td>
<td>237, 335, 370, 1562</td>
<td></td>
</tr>
<tr>
<td>condition</td>
<td>32, 141, 141, 142, 144, 146, 148, 185, 418, 1553</td>
<td></td>
</tr>
<tr>
<td>conditional-expression</td>
<td>132, 226, 444, 1553</td>
<td></td>
</tr>
<tr>
<td>conditionally-supported-directive</td>
<td>427, 428, 1564</td>
<td></td>
</tr>
<tr>
<td>constant-expression</td>
<td>3, 38, 70, 120, 135, 135, 139, 144, 152, 155, 175, 205, 206, 223, 247, 265, 308, 423, 1553</td>
<td></td>
</tr>
<tr>
<td>constraint-expression</td>
<td>91, 105, 139, 169, 336, 349, 349, 350, 449, 1563</td>
<td></td>
</tr>
<tr>
<td>constraint-logical-and-expression</td>
<td>335, 1562</td>
<td></td>
</tr>
<tr>
<td>constraint-logical-or-expression</td>
<td>139, 169, 335, 336, 1562</td>
<td></td>
</tr>
<tr>
<td>contract-attribute-specifier</td>
<td>225, 225, 227, 1560</td>
<td></td>
</tr>
<tr>
<td>contract-level</td>
<td>225, 227, 1560</td>
<td></td>
</tr>
<tr>
<td>control-line</td>
<td>427, 1564</td>
<td></td>
</tr>
<tr>
<td>conversion-declarator</td>
<td>262, 1561</td>
<td></td>
</tr>
<tr>
<td>conversion-function-id</td>
<td>24, 42, 47, 92, 93, 165, 262, 263, 336, 381, 1561, 1588</td>
<td></td>
</tr>
<tr>
<td>conversion-type-id</td>
<td>42, 47, 93, 262, 1561</td>
<td></td>
</tr>
<tr>
<td>coroutine-return-statement</td>
<td>148, 201, 1554</td>
<td></td>
</tr>
</tbody>
</table>
Index of grammar productions
Index of library headers

<table>
<thead>
<tr>
<th>Library Header</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;algorithm&gt;</td>
<td>991</td>
</tr>
<tr>
<td>&lt;any&gt;</td>
<td>586</td>
</tr>
<tr>
<td>&lt;array&gt;</td>
<td>796</td>
</tr>
<tr>
<td>&lt;assert.h&gt;</td>
<td>456, 533, 1592</td>
</tr>
<tr>
<td>&lt;atomic&gt;</td>
<td>1477</td>
</tr>
<tr>
<td>&lt;bit&gt;</td>
<td>1103</td>
</tr>
<tr>
<td>&lt;bitset&gt;</td>
<td>591</td>
</tr>
<tr>
<td>&lt;cassert&gt;</td>
<td>456, 533, 1309, 1592</td>
</tr>
<tr>
<td>&lt;cctype&gt;</td>
<td>755</td>
</tr>
<tr>
<td>&lt;cerrno&gt;</td>
<td>467, 533</td>
</tr>
<tr>
<td>&lt;cfenv&gt;</td>
<td>1094</td>
</tr>
<tr>
<td>&lt;charconv&gt;</td>
<td>711</td>
</tr>
<tr>
<td>&lt;chrono&gt;</td>
<td>1177</td>
</tr>
<tr>
<td>&lt;cinttypes&gt;</td>
<td>1435, 1436</td>
</tr>
<tr>
<td>&lt;climits&gt;</td>
<td>1598</td>
</tr>
<tr>
<td>&lt;clocale&gt;</td>
<td>451, 1592</td>
</tr>
<tr>
<td>&lt;cmath&gt;</td>
<td>1161, 1170</td>
</tr>
<tr>
<td>&lt;codecvt&gt;</td>
<td>1607</td>
</tr>
<tr>
<td>&lt;compare&gt;</td>
<td>129, 503</td>
</tr>
<tr>
<td>&lt;complex&gt;</td>
<td>1095</td>
</tr>
<tr>
<td>&lt;complex.h&gt;</td>
<td>1591, 1595</td>
</tr>
<tr>
<td>&lt;condition_variable&gt;</td>
<td>1525</td>
</tr>
<tr>
<td>&lt;csetjmp&gt;</td>
<td>467, 515, 516, 1593</td>
</tr>
<tr>
<td>&lt;csignal&gt;</td>
<td>515, 516</td>
</tr>
<tr>
<td>&lt;cstdarg&gt;</td>
<td>177, 467, 515</td>
</tr>
<tr>
<td>&lt;cstddef&gt;</td>
<td>118, 127, 1272, 1592, 1593</td>
</tr>
<tr>
<td>&lt;cstdint&gt;</td>
<td>487</td>
</tr>
<tr>
<td>&lt;cstdlib&gt;</td>
<td>110x346</td>
</tr>
<tr>
<td>&lt;cstring&gt;</td>
<td>452, 756, 1592, 1598, 1602</td>
</tr>
<tr>
<td>&lt;ctime&gt;</td>
<td>515, 1269, 1272, 1592</td>
</tr>
<tr>
<td>&lt;ctype.h&gt;</td>
<td>755</td>
</tr>
<tr>
<td>&lt;uchar&gt;</td>
<td>467, 758, 1592</td>
</tr>
<tr>
<td>&lt;wchar.h&gt;</td>
<td>467, 756, 758, 1592</td>
</tr>
<tr>
<td>&lt;cwctype&gt;</td>
<td>467, 755</td>
</tr>
<tr>
<td>&lt;deque&gt;</td>
<td>796</td>
</tr>
<tr>
<td>&lt;errno.h&gt;</td>
<td>533</td>
</tr>
<tr>
<td>&lt;exception&gt;</td>
<td>490</td>
</tr>
<tr>
<td>&lt;execution&gt;</td>
<td>709</td>
</tr>
<tr>
<td>&lt;filesystem&gt;</td>
<td>1390</td>
</tr>
<tr>
<td>&lt;forward_list&gt;</td>
<td>797</td>
</tr>
<tr>
<td>&lt;fstream&gt;</td>
<td>1371</td>
</tr>
<tr>
<td>&lt;functional&gt;</td>
<td>658</td>
</tr>
<tr>
<td>&lt;future&gt;</td>
<td>1531</td>
</tr>
<tr>
<td>&lt;initializer_list&gt;</td>
<td>502</td>
</tr>
<tr>
<td>&lt;inttypes.h&gt;</td>
<td>1436</td>
</tr>
<tr>
<td>&lt;iostream&gt;</td>
<td>1340</td>
</tr>
<tr>
<td>&lt;ios&gt;</td>
<td>1315</td>
</tr>
<tr>
<td>&lt;iosfwd&gt;</td>
<td>1312</td>
</tr>
<tr>
<td>&lt;iostream&gt;</td>
<td>1314</td>
</tr>
<tr>
<td>&lt;iso646.h&gt;</td>
<td>1592, 1595</td>
</tr>
<tr>
<td>&lt;istream&gt;</td>
<td>1339</td>
</tr>
<tr>
<td>&lt;iterator&gt;</td>
<td>878, 1604</td>
</tr>
<tr>
<td>&lt;limits&gt;</td>
<td>479</td>
</tr>
<tr>
<td>&lt;list&gt;</td>
<td>797</td>
</tr>
<tr>
<td>&lt;locale&gt;</td>
<td>1271, 1272</td>
</tr>
<tr>
<td>&lt;locale.h&gt;</td>
<td>1309</td>
</tr>
<tr>
<td>&lt;map&gt;</td>
<td>823</td>
</tr>
<tr>
<td>&lt;memory&gt;</td>
<td>597</td>
</tr>
<tr>
<td>&lt;memory_resource&gt;</td>
<td>645</td>
</tr>
<tr>
<td>&lt;mutex&gt;</td>
<td>1507</td>
</tr>
<tr>
<td>&lt;new&gt;</td>
<td>490</td>
</tr>
<tr>
<td>&lt;numeric&gt;</td>
<td>1080</td>
</tr>
<tr>
<td>&lt;optional&gt;</td>
<td>563</td>
</tr>
<tr>
<td>&lt;ostream&gt;</td>
<td>1339</td>
</tr>
<tr>
<td>&lt;queue&gt;</td>
<td>864</td>
</tr>
<tr>
<td>&lt;random&gt;</td>
<td>1105</td>
</tr>
<tr>
<td>&lt;ratio&gt;</td>
<td>706</td>
</tr>
<tr>
<td>&lt;regex&gt;</td>
<td>1440</td>
</tr>
<tr>
<td>&lt;scoped_allocator&gt;</td>
<td>654</td>
</tr>
<tr>
<td>&lt;set&gt;</td>
<td>824</td>
</tr>
<tr>
<td>&lt;setjmp.h&gt;</td>
<td>516</td>
</tr>
<tr>
<td>&lt;shared_mutex&gt;</td>
<td>1507</td>
</tr>
<tr>
<td>&lt;signal.h&gt;</td>
<td>516</td>
</tr>
<tr>
<td>&lt;span&gt;</td>
<td>872</td>
</tr>
<tr>
<td>&lt;sstream&gt;</td>
<td>1362</td>
</tr>
<tr>
<td>&lt;stack&gt;</td>
<td>865</td>
</tr>
<tr>
<td>&lt;stdalign.h&gt;</td>
<td>1592, 1595</td>
</tr>
<tr>
<td>&lt;stdarg.h&gt;</td>
<td>516</td>
</tr>
<tr>
<td>&lt;stdbool.h&gt;</td>
<td>1592, 1595</td>
</tr>
<tr>
<td>&lt;stddef.h&gt;</td>
<td>18, 21</td>
</tr>
<tr>
<td>&lt;stdexcept&gt;</td>
<td>530</td>
</tr>
<tr>
<td>&lt;stdio.h&gt;</td>
<td>1435</td>
</tr>
<tr>
<td>&lt;stdlib.h&gt;</td>
<td>1596</td>
</tr>
<tr>
<td>&lt;streambuf&gt;</td>
<td>1331</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>719</td>
</tr>
<tr>
<td>&lt;string.h&gt;</td>
<td>756</td>
</tr>
<tr>
<td>&lt;string_view&gt;</td>
<td>746</td>
</tr>
<tr>
<td>&lt;strstream&gt;</td>
<td>1596</td>
</tr>
<tr>
<td>&lt;syncstream&gt;</td>
<td>1384</td>
</tr>
<tr>
<td>&lt;system_error&gt;</td>
<td>535</td>
</tr>
<tr>
<td>&lt;tgmath.h&gt;</td>
<td>1591, 1595</td>
</tr>
<tr>
<td>&lt;thread&gt;</td>
<td>1502</td>
</tr>
<tr>
<td>&lt;time.h&gt;</td>
<td>1269</td>
</tr>
<tr>
<td>&lt;tuple&gt;</td>
<td>553</td>
</tr>
<tr>
<td>Header</td>
<td>Page Numbers</td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
</tr>
<tr>
<td>&lt;type_traits&gt;</td>
<td>682, 1604</td>
</tr>
<tr>
<td>&lt;typeindex&gt;</td>
<td>708</td>
</tr>
<tr>
<td>&lt;typeinfo&gt;</td>
<td>496</td>
</tr>
<tr>
<td>&lt;uchar.h&gt;</td>
<td>758</td>
</tr>
<tr>
<td>&lt;unordered_map&gt;</td>
<td>841</td>
</tr>
<tr>
<td>&lt;unordered_set&gt;</td>
<td>842</td>
</tr>
<tr>
<td>&lt;utility&gt;</td>
<td>544, 1596</td>
</tr>
<tr>
<td>&lt;valarray&gt;</td>
<td>1142</td>
</tr>
<tr>
<td>&lt;variant&gt;</td>
<td>575</td>
</tr>
<tr>
<td>&lt;vector&gt;</td>
<td>798</td>
</tr>
<tr>
<td>&lt;wchar.h&gt;</td>
<td>758</td>
</tr>
<tr>
<td>&lt;wctype.h&gt;</td>
<td>756</td>
</tr>
</tbody>
</table>
Index of library names

complex
  - literals, 1103
  _Exit, 474, 488
  _IOFBF, 1434
  _IOLBF, 1434
  _IONBF, 1434
  __alignas_is_defined, 1595
  __bool_true_false_are_defined, 1595
  __cpp_lib_addressof_constexpr, 477
  __cpp_lib_allocator_traits_is_always_equal, 477
  __cpp_lib_any, 477
  __cpp_lib_apply, 477
  __cpp_lib_array_constexpr, 477
  __cpp_lib_as_const, 477
  __cpp_lib_atomic_is_always_lock_free, 477
  __cpp_lib_atomic_ref, 477
  __cpp_lib_bind_front, 477
  __cpp_lib_bit_cast, 477
  __cpp_lib_bool_constant, 477
  __cpp_lib_bounded_array_traits, 478
  __cpp_lib_boyer_moore_searcher, 478
  __cpp_lib_byte, 478
  __cpp_lib_char8_t, 478
  __cpp_lib_chrono, 478
  __cpp_lib_chrono_udls, 478
  __cpp_lib_clamp, 478
  __cpp_lib_complex_udls, 478
  __cpp_lib_concepts, 478
  __cpp_lib_constexpr_misc, 478
  __cpp_lib_constexpr_swap_algorithms, 478
  __cpp_lib_destroying_delete, 478
  __cpp_lib_enable_shared_from_this, 478
  __cpp_lib_erase_if, 478
  __cpp_lib_exchange_function, 478
  __cpp_lib_execution, 478
  __cpp_lib_filesystem, 478
  __cpp_lib_gcd_lcm, 478
  __cpp_lib_generic_associative_lookup, 478
  __cpp_lib_generic_unordered_hash_lookup, 478
  __cpp_lib_generic_unordered_lookup, 478
  __cpp_lib_generic_unordered_hash_lookup, 478
  __cpp_lib_generic_unordered_lookup, 478
  __cpp_lib_hardware_interference_size, 478
  __cpp_lib_has_unique_object_representations, 478
  __cpp_lib_hypot, 478
  __cpp_lib_incomplete_container_elements, 478
  __cpp_lib_integer_sequence, 478
  __cpp_lib_integral_constant_callable, 478
  __cpp_lib_interpolate, 478
  __cpp_lib_invoke, 478
  __cpp_lib_is_aggregate, 478
  __cpp_lib_is_constant_evaluated, 478
  __cpp_lib_is_final, 478
  __cpp_lib_is_invocable, 478
  __cpp_lib_is_null_pointer, 478
  __cpp_lib_is_swappable, 478
  __cpp_lib_launder, 478
  __cpp_lib_list_remove_return_type, 478
  __cpp_lib_logical_traits, 478
  __cpp_lib_make_from_tuple, 478
  __cpp_lib_make_reverse_iterator, 478
  __cpp_lib_make_unique, 478
  __cpp_lib_map_try_emplace, 478
  __cpp_lib_math_special_functions, 479
  __cpp_lib_memory_resource, 479
  __cpp_lib_node_extract, 479
  __cpp_lib_nonmember_container_access, 479
  __cpp_lib_not_fn, 479
  __cpp_lib_null_iterators, 479
  __cpp_lib_optional, 479
  __cpp_lib_parallel_algorithm, 479
  __cpp_lib_quoted_string_io, 479
  __cpp_lib_ranges, 479
  __cpp_lib_raw_memory_algorithms, 479
  __cpp_lib_result_of_sfinae, 479
  __cpp_lib_robust_nonmodifying_seq_ops, 479
  __cpp_lib_sample, 479
  __cpp_lib_scoped_lock, 479
  __cpp_lib_shared_mutex, 479
  __cpp_lib_shared_ptr_arrays, 479
  __cpp_lib_shared_ptr_weak_type, 479
  __cpp_lib_shared_timed_mutex, 479
  __cpp_lib_string_udls, 479
  __cpp_lib_string_view, 479
  __cpp_lib_three_way_comparison, 479
  __cpp_lib_transformation_trait_aliases, 479
  __cpp_lib_transparent_operators, 479
  __cpp_lib_tuple_element_t, 479
  __cpp_lib_tuples_by_type, 479
  __cpp_lib_type_trait_variable_templates, 479
  __cpp_lib_uncaught_exceptions, 479
  __cpp_lib_unordered_map_try_emplace, 479
  __cpp_lib_variant, 479
  __cpp_lib_void_t, 479

1, 674

a
  cauchy_distribution, 1136
  extreme_value_distribution, 1133
uniform_int_distribution, 1126
uniform_real_distribution, 1127
weibull_distribution, 1132
abbrev
  sys_info, 1254
abort, 80, 147, 455, 474, 488, 495, 500
abs, 474, 1161, 1170, 1435
complex, 1101
duration, 1200
valarray, 1154
absolute, 1422
accumulate, 1084
acos, 1161
complex, 1101
valarray, 1154
acosf, 1161
acosh, 1161
complex, 1102
acoshf, 1161
acoshl, 1161
acosl, 1161
acq_rel
  memory_order, 1481
acquire
  memory_order, 1481
add_const, 699
add_cv, 699
add_lvalue_reference, 699
add_pointer, 700
add_rvalue_reference, 699
add_volatile, 699
address
  coroutine_handle, 513
  coroutine_handle<noop_coroutine_promise>, 515
addressof, 613
adjacent_difference, 1091
adjacent_find, 1032
adopt_lock, 1515
adopt_lock_t, 1515
advance, 904, 905
  subrange, 950
align, 606
align_val_t, 490
aligned_alloc, 474, 618, 1593
aligned_storage, 701, 702
aligned_union, 701
alignment_of, 696
all
  bitset, 595
all_of, 1027
allocate
  allocator, 611
  allocator_traits, 610
memory_resource, 646
polymorphic_allocator, 648
scoped_allocator_adaptor, 657
allocate_bytes
  polymorphic_allocator, 648
allocate_object
  polymorphic_allocator, 648
allocate_shared, 632, 634, 635
allocator, 611
  allocate, 611
deallocate, 611
difference_type, 611
is_always_equal, 611
operator!=, 611
operator=, 611
operator==, 611
propagate_on_container_move_assignment, 611
size_type, 611
value_type, 611
allocator_arg, 606
allocator_arg_t, 606
allocate, 609
  const_pointer, 609
calloc, 609
destruct, 610
destroy, 610
difference_type, 610
is_always_equal, 610
max_size, 611
pointer, 609
propagate_on_container_copy_assignment, 610
propagate_on_container_move_assignment, 610
propagate_on_container_swap, 610
rebind_alloc, 610
select_on_container_copy_construction, 611
size_type, 610
void_pointer, 610
allocator_type
  basic_string, 723
alpha
gamma_distribution, 1132
always_noconv
codecvt, 1285
ambiguous
  local_info, 1255
ambiguous_local_time, 1253
constructor, 1253
any
  constructor, 587, 588
destructor, 588
emplace, 589
has_value, 590
operator=, 588, 589
reset, 589
compare_exchange_strong, 1484
compare_exchange_weak, 1484
constructor, 1484
exchange, 1484
is_always_lock_free, 1483
is_lock_free, 1484
load, 1484
operator type, 1484
operator=, 1484
required_alignment, 1484
store, 1484
value_type, 1483
atomic_ref<floating-point>, 1486
atomic_ref<integral>, 1485
atomic_ref<T*>, 1487
atomic_schar, 1481
atomic_short, 1481
ATOMIC_SHORT_LOCK_FREE, 1482
atomic_signal_fence, 1499
atomic_size_t, 1481
atomic_store, 1490
shared_ptr, 1606
atomic_store_explicit, 1490
atomic_thread_fence, 1498
atomic_uchar, 1481
atomic_uint, 1481
atomic_uint16_t, 1481
atomic_uint32_t, 1481
atomic_uint64_t, 1481
atomic_uint8_t, 1481
atomic_uint_fast16_t, 1481
atomic_uint_fast32_t, 1481
atomic_uint_fast64_t, 1481
atomic_uint_fast8_t, 1481
atomic_uint_least16_t, 1481
atomic_uint_least32_t, 1481
atomic_uint_least64_t, 1481
atomic_uint_least8_t, 1481
atomic_uintmax_t, 1481
atomic_ullong, 1481
atomic_ulong, 1481
atomic_ushort, 1481
ATOMIC_VAR_INIT, 1490
atomic_wchar_t, 1481
ATOMIC_WCHAR_T_LOCK_FREE, 1482
auto_ptr
  zombie, 465
auto_ptr_ref
  zombie, 465
await_ready
  suspend_always, 515
  suspend_never, 515
await_resume
  suspend_always, 515
  suspend_never, 515
await_suspend
  suspend_always, 515
  suspend_never, 515
b
  cauchy_distribution, 1136
extreme_value_distribution, 1133
uniform_int_distribution, 1126
uniform_real_distribution, 1127
weibull_distribution, 1132

back
basic_string, 731
basic_string_view, 750
span, 876

back_insert_iterator, 911
constructor, 911
operator*, 911
operator++, 911
operator=, 911

back_inserter, 911

bad
basic_ios, 1328
bad_alloc, 122, 491, 494, 495
constructor, 494
what, 495

bad_any_cast, 586
what, 586

bad_array_new_length, 495
constructor, 495
what, 495

bad_cast, 110, 496, 497
constructor, 497
what, 497

bad_exception, 500
constructor, 500
what, 500

bad_function_call, 674
constructor, 674
what, 674

bad_optional_access
constructor, 572
what, 572

bad_typeid, 111, 496, 498
constructor, 498
what, 498

bad_variant_access, 585
constructor, 585
what, 586

bad_weak_ptr, 627
constructor, 627
what, 627

base
common_view, 984
counted_iterator, 924
filter_view, 960
filter_view::iterator, 962
filter_view::sentinel, 963
move_iterator, 915
reverse_iterator, 908
reverse_view, 985
take_view, 970
take_view::sentinel, 971
transform_view, 964
transform_view::iterator, 966
transform_view::sentinel, 968

basic_common_reference, 701

basic_filebuf, 1312, 1372
close, 1375
constructor, 1373
destructor, 1374
imbue, 1377
is_open, 1374
open, 1374
operator=, 1374
overflow, 1376
pbackfail, 1376
seekoff, 1377
seekpos, 1377
setbuf, 1376
showmanyc, 1375
swap, 1374
sync, 1377
uflow, 1376
underflow, 1375

basic_filebuf<char>, 1371
basic_filebuf<wchar_t>, 1371

basic_fstream, 1312, 1382
close, 1384
constructor, 1383
is_open, 1383
open, 1383, 1384
operator=, 1383
rdbuf, 1383
swap, 1383

basic_fstream<char>, 1371
basic_fstream<wchar_t>, 1371

basic_ifstream, 1312, 1378
close, 1379
constructor, 1378, 1379
is_open, 1379
open, 1379
operator=, 1379
rdbuf, 1379
swap, 1379

basic_ifstream<char>, 1371
basic_ifstream<wchar_t>, 1371

basic_ios, 1312, 1325
bad, 1328
clear, 1328
constructor, 1326
copyfmt, 1327
destructor, 1326
eof, 1328
exceptions, 1328
fail, 1328
fill, 1327
good, 1328
imbue, 1326
init, 1326, 1342
move, 1327
narrow, 1327
operator bool, 1328
operator!, 1328
rdbuf, 1326
Index of library names 1685
pubsync, 1334
sbumpc, 1335
seekoff, 1336
seekpos, 1336
setbuf, 1336, 1367
setg, 1336
setp, 1336
ggetc, 1335
gsgetc, 1335
showmanyc, 1337
stxgetc, 1334
putbckc, 1335
putc, 1335
putn, 1335
sungetc, 1335
swap, 1335
sync, 1337
uflow, 1338
underflow, 1337
xsgetn, 1337
xsputn, 1338

basic_streambuf<char>, 1331
basic_streambuf<wchar_t>, 1331
basic_string, 723, 740, 1363, 1607
allocator_type, 723
append, 732, 733
assign, 733, 734
at, 731
back, 731
begin, 730
c_str, 738
capacity, 730
cbegin, 730
cend, 730
clear, 731
compare, 739, 740
const_iterator, 723
const_pointer, 723
const_reference, 723
const_reverse_iterator, 723
constructor, 728, 729
copy, 737
crbegin, 730
crend, 730
data, 738
difference_type, 723
empty, 731
end, 730
ends_with, 740
erase, 735
find, 738
find_first_not_of, 738
find_first_of, 738
find_last_not_of, 738
find_last_of, 738
front, 731
get_allocator, 738
ggetline, 743
insert, 734, 735
iterator, 723
length, 730
max_size, 730
operator basic_string_view, 738
operator*, 740, 741
operator*=, 731, 732
operator<<, 743
operator=, 729, 730
operator>>, 742
operator[], 731
pointer, 723
pop_back, 736
push_back, 733
rbegin, 730
reference, 723
rend, 730
replace, 736, 737
reserve, 731, 1607
resize, 730
reverse_iterator, 723
rfind, 738
shrink_to_fit, 731
size, 730
size_type, 723
starts_with, 740
substr, 739
swap, 737, 742
traits_type, 723
value_type, 723

basic_string_view, 747
at, 750
back, 750
begin, 749
cbegin, 749
cend, 749
compare, 751
const_iterator, 747, 749
const_pointer, 747
const_reference, 747
const_reverse_iterator, 747
constructor, 749
copy, 751
crbegin, 749
crend, 749
data, 750
difference_type, 747
empty, 750
end, 749
ends_with, 752
find, 752
find_first_not_of, 753
find_first_of, 752
find_last_not_of, 753
find_last_of, 753
front, 750
iterator, 747
length, 750
max_size, 750
operator!, 754
operator<, 754
operator<<, 754
operator=, 754
operator>, 754
operator[], 750
pointer, 747
rbegin, 749
reference, 747
remove_prefix, 750
remove_suffix, 750
rend, 749
reverse_iterator, 747
rfind, 752
size, 750
size_type, 747
starts_with, 751
substr, 751
swap, 750
traits_type, 747
value_type, 747
basic_stringbuf, 1312, 1363
constructor, 1364
operator=, 1365
overflow, 1366
pbackfail, 1365
seekoff, 1366
seekpos, 1367
str, 1365
swap, 1365
underflow, 1365
basic_stringbuf<char>, 1362
basic_stringbuf<wchar_t>, 1362
basic_stringstream, 1312, 1370
constructor, 1371
operator=, 1371
rdbuf, 1371
str, 1371
swap, 1371
basic_stringstream<char>, 1362
basic_stringstream<wchar_t>, 1362
basic_syncbuf, 1312, 1384
constructor, 1385
destructor, 1385
emit, 1386
get_allocator, 1386
get_wrapped, 1386
operator=, 1385
set_emit_on_sync, 1386
swap, 1386
sync, 1386
before
	type_info, 497
before_begin
	forward_list, 807
begin, 502
array, 799
basic_string, 730
basic_string_view, 749
directory_iterator, 1419
filter_view, 960
initializer_list, 503
iota_view, 954
match_results, 1463
path, 1408
recursive_directory_iterator, 1422
reverse_view, 985
single_view, 952
span, 876
split_view::outer_iterator::value_type,
980
subrange, 949
sys_info, 1254
transform_view, 964
tzdb_list, 1251
unordered associative containers, 794
valarray, 1161
begin(C&), 934
begin(initializer_list<E>), 503
begin(T (&)[N]), 934
bernoulli_distribution, 1127
constructor, 1127
p, 1127
beta, 1171
gamma_distribution, 1132
betal, 1171
betaf, 1171
bind_front
bidirectional_iterator_tag, 903
BidirectionalIterator, 895
BidirectionalRange, 944
big
	enian, 705
binary_function
zombie, 465
binary_negate
zombie, 465
binary_search, 1061
bind, 673–674
bind1st
zombie, 465
bind2nd
zombie, 465
bind_front, 672
binder1st
zombie, 466
binder2nd
zombie, 466
binomial_distribution, 1127
constructor, 1128
p, 1128
t, 1128
bit_and, 670
operator(), 670
bit_and<>, 670
operator(), 670
bit_cast, 1104
Index of library names 1687
Index of library names

bit_not
operator(), 671
bit_not<>, 671
operator(), 671
bit_or, 670
operator(), 670
bit_or<>, 670
operator(), 671
bit_xor, 671
operator(), 671
bit_xor<>, 671
operator(), 671
bitset, 591
all, 595
any, 595
constructor, 593
count, 595
flip, 594
none, 595
operator!=, 595
operator>>, 595, 596
operator<<, 594
operator<<=, 594
operator>>, 594
operator>>=, 594
operator|, 596
operator|, 594
reset, 594
set, 594
c16rtomb, 758
c32rtomb, 758
c8rtomb, 758

c_str
basic_string, 738
path, 1403
cacos
complex, 1101
cacosh
complex, 1102
call_once, 1524
calloc, 474, 618, 1593
canonical, 1422
capacity
basic_string, 730
vector, 819
casin
complex, 1102
casinh
complex, 1102
catan
complex, 1102
catanh
complex, 1102
cATEGORY
error_code, 540
eerror_condition, 541
locale, 1274
canonical_distribution, 1136
cauchy_distribution, 1136
cbegin(const C&), 934
cbegin
forward_list, 807
bsearch, 474, 1093
btowc, 756
bucket
unordered associative containers, 793
cbucket_count
unordered associative containers, 793

cbucket_size
unordered associative containers, 793
BUFSIZE, 1434
byte, 473
common_type, 702, 1194, 1197
common_view
    common_view, 983
CommonRange, 945
CommonReference, 522
comp_ellint_1, 1171
comp_ellint_1f, 1171
comp_ellint_1l, 1171
comp_ellint_2, 1172
comp_ellint_2f, 1172
comp_ellint_2l, 1172
comp_ellint_3, 1172
comp_ellint_3f, 1172
comp_ellint_3l, 1172
cmp_3way, 1079
cmp_exchange_strong
    atomic, 1490
    atomic<floating-point>, 1490
    atomic<integral>, 1490
    atomic<shared_ptr<T>>, 643
    atomic<T>, 1490
    atomic<weak_ptr<T>>, 645
    atomic_ref, 1484
    atomic_ref<floating-point>, 1484
    atomic_ref<integral>, 1484
    atomic_ref<T>, 1484
cmp_exchange_weak
    atomic, 1490
    atomic<floating-point>, 1490
    atomic<integral>, 1490
    atomic<shared_ptr<T>>, 643
    atomic<T>, 1490
    atomic<weak_ptr<T>>, 645
    atomic_ref, 1484
    atomic_ref<floating-point>, 1484
    atomic_ref<integral>, 1484
    atomic_ref<T>, 1484
cmplx, 1097
    constructor, 1099
    imag, 1099
    operator!=, 1100
    operator>>, 1100
    real, 1099
    value_type, 1097
    concat
    path, 1402
    <concepts>, 519
    condition_variable, 1526
        constructor, 1526
        destructor, 1527
        notify_all, 1527
        notify_one, 1527
        wait, 1527
        wait_for, 1528, 1529
        wait_until, 1527, 1528
    condition_variable_any, 1529
        constructor, 1529
        destructor, 1530
        notify_all, 1530
        notify_one, 1530
        wait, 1530
        wait_for, 1531
        wait_until, 1530, 1531
    conj, 1103
    complex, 1101
    conjunction, 704
    const_iterator
        basic_string, 723
        basic_string_view, 747, 749
    const_local_iterator
        unordered associative containers, 785
    const_mem_fun1_ref_t
        zombie, 466
    const_mem_fun1_t
        zombie, 466
    const_mem_fun_ref_t
        zombie, 466
    const_mem_fun_t
        zombie, 466
    const_pointer
        allocator_traits, 609
        basic_string, 723
        basic_string_view, 747
        scoped_allocator_adaptor, 655
    const_pointer_cast
        shared_ptr, 637
    const_reference
        basic_string, 723
        basic_string_view, 747
    const_reverse_iterator
        basic_string, 723
        basic_string_view, 747
    const_void_pointer
        allocator_traits, 610
        scoped_allocator_adaptor, 655
    construct
        allocator_traits, 610
        polymorphic_traits, 649
        scoped_allocator_adaptor, 657
Constructible, 525
consume
memory_order, 1481
contains
ordered associative containers, 781
unordered associative containers, 792
contiguous_iterator_tag, 903
ContiguousIterator, 896
ContiguousRange, 945
contract_violation, 498
assertion_level, 499
comment, 498
file_name, 498
function_name, 498
line_number, 498
converted
wstring_convert, 1610
ConvertibleTo, 521
copy, 1038, 1039
basic_string, 737
basic_string_view, 751
path, 1423
copy_backward, 1040
copy_file, 1424
copy_if, 1039
copy_n, 1039
copy_options, 1411
copy_symlink, 1425
Copyable, 528
CopyConstructible, 525
copyfmt
basic_ios, 1327
copysign, 1161
copysignf, 1161
copysignl, 1161
<coroutine>, 511
coroutine_handle, 512
address, 513
constructor, 513
destroy, 514
done, 513
from_address, 513
from_promise, 513
operator bool, 514
operator()?, 515
promise, 515
resume, 515
cos, 1161
complex, 1102
valarray, 1154
cosf, 1161
cosh, 1161
complex, 1102
valarray, 1154
coshf, 1161
coshl, 1161
cosl, 1161
count, 1032
counted_iterator, 922
copy_backward, 1040
copy_file, 1424
copy_if, 1039
copy_n, 1039
copy_options, 1411
copy_symlink, 1425
Copyable, 528
CopyConstructible, 525
copyfmt
basic_ios, 1327
copysign, 1161
copysignf, 1161
copysignl, 1161
<coroutine>, 511
coroutine_handle, 512
address, 513
constructor, 513
destroy, 514
done, 513
from_address, 513
from_promise, 513
operator bool, 513
operator!?, 514
operator(), 513
operator<, 514
operator=, 514
operator!=, 514
operator<, 514
operator<=, 514
operator==, 514
operator>=, 514
operator++, 514
operator+=, 514
operator++, 514
operator+=, 514
operator<, 514
operator<, 514
operator<=, 514
operator<=, 514
operator==, 514
operator==, 514
operator>=, 514
operator>=, 514
promise, 515
resume, 515
coroutine_handle<noop_coroutine_promise>, 514
address, 515
destroy, 515
done, 515
coroutine_handle<noop_coroutine_promise>, 514
address, 515
destroy, 515
done, 515
coroutine_handle<noop_coroutine_promise>, 514
address, 515
destroy, 515
done, 515
crend(const C& c), 935

cshift
valarray, 1151
cbind, 1269
ctype, 1279
do_is, 1280
do_narrow, 1281
do_scan_not, 1281
do_tolower, 1281
do_toupper, 1281
do_widen, 1281
is, 1280
narrow, 1280
scan_is, 1280
scan_not, 1280
tolower, 1280
toupper, 1280
widen, 1280
cctype<char>, 1282
c classic_table, 1284
c constructor, 1283
c cctype<char>, 1283
c destructor, 1283
c do_narrow, 1284
c do_tolower, 1284
c do_toupper, 1284
c do_widen, 1284
is, 1283
narrow, 1283
scan_is, 1283
scan_not, 1283
table, 1283
tolower, 1283
toupper, 1283
widen, 1283
cctype_base, 1279
do_scan_is, 1281
c cctype_byname, 1282
c curr_symbol
moneypunct, 1307
c current_exception, 501
c current_path, 1427
c current_zone, 1252
tzdb, 1250
c cv_status, 1525
cyl_bessel_i, 1172
cyl_bessel_if, 1172
cyl_bessel_il, 1172
cyl_bessel_j, 1172
cyl_bessel_jf, 1172
cyl_bessel_jl, 1172
cyl_bessel_k, 1173
cyl_bessel_kf, 1173
cyl_bessel_kl, 1173
cyl_neumann, 1173
cyl_neumannf, 1173
cyl_neumannl, 1173
data array, 800
basic_string, 738
basic_string_view, 750
single_view, 952
span, 876
vector, 820
data(C& c), 935
data(initializer_list<E>), 935
data(T (&array)[N]), 935
date
leap, 1262
date_order
time_get, 1299
day, 1217
c constructor, 1217
from_stream, 1218
month_day, 1227
ok, 1218
operator unsigned, 1218
operator"d, 1218
operator+, 1218
operator++, 1217
operator+=, 1217
operator-, 1218
operator-=, 1217
operator==, 1217
to_stream, 1218
year_month_day, 1234
year_month_day_last, 1237
days, 1177
DBL_DECIMAL_DIG, 487
DBL_DIG, 487
DBL_EPSILON, 487
DBL_HAS_SUBNORM, 487
DBL_MANT_DIG, 487
DBL_MAX, 487
DBL_MAX_10_EXP, 487
DBL_MAX_EXP, 487
DBL_MIN, 487
DBL_MIN_10_EXP, 487
DBL_MIN_EXP, 487
DBL_TRUE_MIN, 487
deallocate
 allocator, 611
 allocator_traits, 610
 memory_resource, 646
 polymorphic_allocator, 648
 scoped_allocator_adaptor, 657
deallocate_bytes
 polymorphic_allocator, 648
deallocate_object
 polymorphic_allocator, 649
dec, 1330, 1355
decay, 701
DECIMAL_DIG, 487
decimal_point
moneypunct, 1307
numpunct, 1295
declare_no_pointers, 605
declare_reachable, 605
decval, 548
default_delete
constructor, 619
operator(), 619
default_error_condition
error_category, 537, 538
error_code, 540
default_random_engine, 1122
default_searcher, 678
constructor, 679
operator(), 679
default_sentinel, 878
default_sentinel_t, 922
default_zone
zoned_traits<const time_zone*>, 1257
DefaultConstructible, 525
defaultfloat, 1331
defer_lock, 1515
defer_lock_t, 1515 delete
operator, 468, 491–494, 618
denormAbsent, 480
denormIndeterminate, 480
denormMin
numeric_limits, 484
denormPresent, 480
densities
piecewiseConstantDistribution, 1140
piecewiseLinearDistribution, 1142
depth
recursive_directory_iterator, 1421
ddeque, 801
constructor, 803
shrink_to_fit, 803
DerivedFrom, 521
destroy, 617
allocator_traits, 610
coroutineHandle, 514
coroutineHandle<noop_coroutine_promise>, 515
polymorphic_allocator, 649
scoped_allocator_adaptor, 658
destroyAt, 617
destroyN, 617, 618
destroying_delete, 490
destroying_delete_t, 490
Destructible, 525
detach
thread, 1506
difference_type
allocator, 611
allocator_traits, 610
basic_string, 723
basic_string_view, 747
pointer_traits, 604
scoped_allocator_adaptor, 655
difftime, 1269
digits
numeric_limits, 482
digits10
numeric_limits, 482
directory_entry, 1413
assign, 1415
constructor, 1415
exists, 1416
file_size, 1416
hard_link_count, 1416
is_block_file, 1416
is_character_file, 1416
is_directory, 1416
is фifo, 1416
is other, 1416
is_regular_file, 1416
is_socket, 1416
is_symlink, 1416
last_write_time, 1417
operator const filesystem::path&, 1415
operator!=, 1417
operator<, 1417
operator<=, 1417
operator==, 1417
operator>, 1417
operator>=, 1417
path, 1415
refresh, 1415
replace_filename, 1415
status, 1417
symlink_status, 1417
directory_iterator, 1417
constructor, 1418, 1419
increment, 1419
operator++, 1419
operator*, 1419
directory_options, 1411
disable_recursion_pending
recursive_directory_iterator, 1422
discard_block_engine, 1118, 1119
constructor, 1119
discrete_distribution, 1138
constructor, 1138, 1139
probabilities, 1139
disjunction, 704
distance, 904–906
div, 474, 1435
divT, 474
divides, 664
operator(), 665
divides<>, 665
operator(), 665
do_allocate
memory_resource, 647
monotonic_buffer_resource, 654
synchronized_pool_resource, 652
unsynchronized_pool_resource, 652
do_always_noconv
codecvt, 1287
do_close
message, 1309
do_compare
collate, 1297
do_curr_symbol
moneypunct, 1307
do_date_order
time_get, 1300
do_deallocate
memory_resource, 647
monotonic_buffer_resource, 654
synchronized_pool_resource, 652
unsynchronized_pool_resource, 652
do_decimal_point
moneypunct, 1307
numpunct, 1296
do_encoding
codecvt, 1287
do_falsename
numpunct, 1296
do_frac_digits
moneypunct, 1307
do_get
messages, 1309
money_get, 1303
num_get, 1289, 1291
time_get, 1301
do_get_date
time_get, 1300
do_get_monthname
time_get, 1300
do_get_time
time_get, 1300
do_get_weekday
time_get, 1300
do_get_year
time_get, 1301
do_grouping
moneypunct, 1307
numpunct, 1296
do_hash
collate, 1297
do_in
codecvt, 1286
do_is
cctype, 1280
do_is_equal
memory_resource, 647
monotonic_buffer_resource, 654
synchronized_pool_resource, 653
unsynchronized_pool_resource, 653
do_length
codecvt, 1287
do_max_length
codecvt, 1287
do_narrow, 1283
cctype, 1281
cctype<char>, 1284
do_neg_format
moneypunct, 1307
do_negative_sign
moneypunct, 1307
do_open
messages, 1308
do_out
codecvt, 1286
do_pos_format
moneypunct, 1307
do_positive_sign
moneypunct, 1307
do_put
money_put, 1305
num_put, 1292, 1294
time_put, 1302
do_scan_is
ccctype_base, 1281
do_scan_not
ccctype, 1281
do_thousands_sep
moneypunct, 1307
numpunct, 1296
do_tolower
ccctype, 1281
ccctype<char>, 1284
do_toupper
ccctype, 1281
ccctype<char>, 1284
do_transform
collate, 1297
do_truename
numpunct, 1296
do_unshift
ccctype, 1287
do_widen
ccctype, 1281
ccctype<char>, 1284
domain_error, 530, 531
constructor, 531
done
coroutine_handle, 513
coroutine_handle noop_coroutine promise>, 514
double_t, 1161
duration, 1194
abs, 1200
cei1, 1199
constructor, 1195, 1196
count, 1196
duration_cast, 1199
floor, 1199
from_stream, 1201
max, 1197
min, 1197
operator!=, 1198
Index of library names
<table>
<thead>
<tr>
<th>enable_shared_from_this</th>
<th>641</th>
</tr>
</thead>
<tbody>
<tr>
<td>constructor, 641</td>
<td></td>
</tr>
<tr>
<td>operator=, 641</td>
<td></td>
</tr>
<tr>
<td>shared_from_this, 641</td>
<td></td>
</tr>
<tr>
<td>weak_from_this, 641</td>
<td></td>
</tr>
<tr>
<td>enable_view, 944</td>
<td></td>
</tr>
<tr>
<td>ENAMETOOLONG, 533</td>
<td></td>
</tr>
<tr>
<td>encoding</td>
<td></td>
</tr>
<tr>
<td>codecvt, 1285</td>
<td></td>
</tr>
<tr>
<td>end(C&amp;)</td>
<td>934</td>
</tr>
<tr>
<td>end(initializer_list&lt;E&gt;)</td>
<td>503</td>
</tr>
<tr>
<td>end(T (&amp;)[N])</td>
<td>934</td>
</tr>
<tr>
<td>endian</td>
<td>705</td>
</tr>
<tr>
<td>big, 705</td>
<td></td>
</tr>
<tr>
<td>little, 705</td>
<td></td>
</tr>
<tr>
<td>native, 705</td>
<td></td>
</tr>
<tr>
<td>endl, 1355, 1357</td>
<td></td>
</tr>
<tr>
<td>ends, 1357</td>
<td></td>
</tr>
<tr>
<td>ends_with</td>
<td></td>
</tr>
<tr>
<td>basic_string, 740</td>
<td></td>
</tr>
<tr>
<td>basic_string_view, 752</td>
<td></td>
</tr>
<tr>
<td>ENETDOWN, 533</td>
<td></td>
</tr>
<tr>
<td>ENETRESET, 533</td>
<td></td>
</tr>
<tr>
<td>ENETUNREACH, 533</td>
<td></td>
</tr>
<tr>
<td>ENFILE, 533</td>
<td></td>
</tr>
<tr>
<td>ENOBUFFS, 533</td>
<td></td>
</tr>
<tr>
<td>ENODATA, 533</td>
<td></td>
</tr>
<tr>
<td>ENODEV, 533</td>
<td></td>
</tr>
<tr>
<td>ENOENT, 533</td>
<td></td>
</tr>
<tr>
<td>ENOEXEC, 533</td>
<td></td>
</tr>
<tr>
<td>ENOLCK, 533</td>
<td></td>
</tr>
<tr>
<td>ENOLINK, 533</td>
<td></td>
</tr>
<tr>
<td>ENOMEM, 533</td>
<td></td>
</tr>
<tr>
<td>ENOMSG, 533</td>
<td></td>
</tr>
<tr>
<td>ENOPROTOOPT, 533</td>
<td></td>
</tr>
<tr>
<td>ENOSPAC, 533</td>
<td></td>
</tr>
<tr>
<td>ENOSR, 533</td>
<td></td>
</tr>
<tr>
<td>ENOSTR, 533</td>
<td></td>
</tr>
<tr>
<td>ENSYS, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTCONN, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTDIR, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTEMPTY, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTRECOVERABLE, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTSOCK, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTSUP, 533</td>
<td></td>
</tr>
<tr>
<td>ENOTTY, 533</td>
<td></td>
</tr>
<tr>
<td>entropy</td>
<td></td>
</tr>
<tr>
<td>random_device, 1123</td>
<td></td>
</tr>
<tr>
<td>ENXIO, 533</td>
<td></td>
</tr>
<tr>
<td>EOF, 1434</td>
<td></td>
</tr>
<tr>
<td>eof</td>
<td></td>
</tr>
<tr>
<td>basic_ios, 1328</td>
<td></td>
</tr>
<tr>
<td>EPROTONOSUPPORT, 533</td>
<td></td>
</tr>
<tr>
<td>EPROTO, 533</td>
<td></td>
</tr>
<tr>
<td>EPROTOTYPE, 533</td>
<td></td>
</tr>
<tr>
<td>epsilon</td>
<td></td>
</tr>
<tr>
<td>numeric_limits, 482</td>
<td></td>
</tr>
<tr>
<td>equal, 1034</td>
<td></td>
</tr>
<tr>
<td>istreambuf_iterator, 932</td>
<td></td>
</tr>
<tr>
<td>strong_equality, 505</td>
<td></td>
</tr>
<tr>
<td>strong_ordering, 508</td>
<td></td>
</tr>
<tr>
<td>equal_range, 1061</td>
<td></td>
</tr>
<tr>
<td>ordered associative containers, 781</td>
<td></td>
</tr>
<tr>
<td>unordered associative containers, 794</td>
<td></td>
</tr>
<tr>
<td>equal_to, 666, 668</td>
<td></td>
</tr>
<tr>
<td>operator(), 666</td>
<td></td>
</tr>
<tr>
<td>equal_to&lt;&gt;(), 666</td>
<td></td>
</tr>
<tr>
<td>operator(), 666</td>
<td></td>
</tr>
<tr>
<td>EqualityComparable, 527</td>
<td></td>
</tr>
<tr>
<td>EqualityComparableWith, 527</td>
<td></td>
</tr>
<tr>
<td>equivalent, 1427</td>
<td></td>
</tr>
<tr>
<td>error_category, 538</td>
<td></td>
</tr>
<tr>
<td>partial_ordering, 506</td>
<td></td>
</tr>
<tr>
<td>strong_equality, 505</td>
<td></td>
</tr>
<tr>
<td>strong_ordering, 508</td>
<td></td>
</tr>
<tr>
<td>weak_equality, 504</td>
<td></td>
</tr>
<tr>
<td>weak_ordering, 507</td>
<td></td>
</tr>
<tr>
<td>ERANGE, 533</td>
<td></td>
</tr>
<tr>
<td>erase</td>
<td></td>
</tr>
<tr>
<td>basic_string, 735, 743</td>
<td></td>
</tr>
<tr>
<td>deque, 804</td>
<td></td>
</tr>
<tr>
<td>forward_list, 810</td>
<td></td>
</tr>
<tr>
<td>list, 814, 816</td>
<td></td>
</tr>
<tr>
<td>ordered associative containers, 780</td>
<td></td>
</tr>
<tr>
<td>unordered associative containers, 791</td>
<td></td>
</tr>
<tr>
<td>vector, 820, 821</td>
<td></td>
</tr>
<tr>
<td>erase_after</td>
<td></td>
</tr>
<tr>
<td>forward_list, 808</td>
<td></td>
</tr>
<tr>
<td>tzdb_list, 1251</td>
<td></td>
</tr>
</tbody>
</table>
erase_if
basic_string, 744
deque, 804
forward_list, 811
list, 816
map, 831
multimap, 834
multiset, 841
set, 837
unordered_map, 849
unordered_multimap, 854
unordered_multiset, 864
unordered_set, 859
vector, 821
operator==, 542
operator==, 542
value, 541
error_type, 1448, 1449
ESPIPE, 533
ESRCH, 533
ETIME, 533
ETIMEDOUT, 533
ETXTBSY, 533
EWOULDBLOCK, 533
exception, 499
constructor, 499
destructor, 499
operator=, 499
what, 499
exception_ptr, 501
exceptions
basic_ios, 1328
exchange, 547
atomic, 1490
atomic<floating-point>, 1490
atomic<integral>, 1490
atomic<shared_ptr<T>>, 643
atomic<T*>, 1490
atomic<weak_ptr<T>>, 645
atomic_ref, 1484
atomic_ref<floating-point>, 1484
atomic_ref<integral>, 1484
atomic_ref<T*>, 1484
exclusive_scan, 1088
EXDEV, 533
execution
par, 711
par_unseq, 711
seq, 711
execution::parallel_policy, 710
execution::parallel_unsequenced_policy, 710
execution::sequenced_policy, 710
execution::unsequenced_policy, 710
exists, 1427
directory_entry, 1416
exit, 77, 79, 147, 455, 474, 489, 495
EXIT_FAILURE, 474
EXIT_SUCCESS, 474
exp, 1161
complex, 1102
valarray, 1154
exp2, 1161
exp2f, 1161
exp2l, 1161
expf, 1161
expint, 1174
expintf, 1174
expintl, 1174
expired
weak_ptr, 640
exp1, 1161
expm1, 1161
expmif, 1161
Index of library names 1697
expm1l, 1161
exponential_distribution, 1130
constructor, 1131
lambda, 1131
extension
path, 1405
extent, 696
extract
ordered associative containers, 779
unordered associative containers, 790
extreme_value_distribution, 1133
a, 1133
b, 1133
constructor, 1133
fabs, 1161
fabsf, 1161
fabsl, 1161
facet
locale, 1275
fail
basic_ios, 1328
failed
ostreambuf_iterator, 934
false_type, 688
false_name
numpunct, 1296
fclose, 1375, 1434
fdim, 1161
fdimf, 1161
fdiml, 1161
FE_ALL_EXCEPT, 1094
FE_DFL_ENV, 1094
FE_DIVBYZERO, 1094
FE_DOWNWARD, 1094
FE_INEXACT, 1094
FE_INVALID, 1094
FE_OVERFLOW, 1094
FE_TONEAREST, 1094
FE_TOWARDZERO, 1094
FE_UPWARD, 1094
fenv_t
filebuf, 1312, 1371
filename
path, 1405
FILENAME_MAX, 1434
filesystem_error, 1409
constructor, 1409
path1, 1410
operator<<, 1212
to_stream, 1212
file_type, 1410
fill, 1046
array, 800
basic_ios, 1327
fill_n, 1046
filter_view
filter_view, 960
find, 1029
atomic_ref<integral>, 1486
atomic_ref<T*>, 1488
fetch_and
atomic<integral>, 1493
atomic_ref<integral>, 1486
fetch_or
atomic<integral>, 1493
atomic_ref<integral>, 1486
fetch_sub
atomic<floating-point>, 1495
atomic<integral>, 1493
atomic<T*>, 1496
atomic_ref<floating-point>, 1487
atomic_ref<integral>, 1486
atomic_ref<T*>, 1488
fetch_xor
atomic<integral>, 1493
atomic_ref<integral>, 1486
fetestexcept, 1094
feupdateenv, 1094
fexcept_t, 1094
fflush, 1434
fgetc, 1434
fgetpos, 1434
fgets, 1434
fgetwc, 756
fgetws, 756
FILE, 1434
file_clock, 1211
now, 1211
file_name
contract_violation, 498
file_size, 1427
directory_entry, 1416
file_status, 1411
constructor, 1413
permissions, 1413
type, 1413
file_time, 1177
from_stream, 1212
operator<<, 1212
to_stream, 1212
file_type, 1410
filebuf, 1312, 1371
filename
path, 1405
FILENAME_MAX, 1434
filesystem_error, 1409
constructor, 1409
path1, 1410
path2, 1410
what, 1410
fill, 1046
array, 800
basic_ios, 1327
fill_n, 1046
filter_view
filter_view, 960
find, 1029
basic_string, 738
basic_string_view, 752
ordered associative containers, 781
unordered associative containers, 791
find_end, 1030
find_first_not_of
basic_string, 738
basic_string_view, 753
find_first_of, 1031
basic_string, 738
basic_string_view, 752
find_if, 1029
find_if_not, 1029
find_last_not_of
basic_string, 738
basic_string_view, 753
find_last_of
basic_string, 738
basic_string_view, 753
first
local_info, 1255
span, 875
first_argument_type
zombie, 466
fisher_f_distribution, 1136
constructor, 1137
m, 1137
n, 1137
fixed, 1330
fixed
chars_format, 711
flag_type
basic_regex, 1455
flags
ios_base, 1279, 1321
flip
bitset, 594
vector<bool>, 822
float_denorm_style, 479, 480
numeric_limits, 483
float_round_style, 479, 480
float_t, 1161
floor, 1161
duration, 1199
time_point, 1204
floor2, 1104
floorf, 1161
floorl, 1161
FLT_DECIMAL_DIG, 487
FLT_DIG, 487
FLT_EPSILON, 487
FLT_EVAL_METHOD, 487
FLT_HAS_SUBNORM, 487
FLT_MANT_DIG, 487
FLT_MAX, 487
FLT_MAX_10_EXP, 487
FLT_MAX_EXP, 487
FLT_MIN, 487
FLT_MIN_10_EXP, 487
FLT_MIN_EXP, 487
FLT_ROUNDS, 487
FLT_TRUE_MIN, 487
flush, 1321, 1343, 1353, 1358
basic_ostream, 1357
flush_emit, 1358
fma, 1161
fmaf, 1161
fmal, 1161
fmax, 1161
fmaxf, 1161
fmaxl, 1161
fmin, 1161
fminf, 1161
fminl, 1161
fmod, 1161
fmodf, 1161
fmodl, 1161
fmtflags
ios_base, 1319, 1359
fopen, 1374, 1434
FOPEN_MAX, 1434
for_each, 1028
for_each_n, 1028, 1029
format, 1263–1266
match_results, 1464
format_default, 1446, 1448
format_first_only, 1446, 1448, 1468
format_no_copy, 1446, 1448, 1468
format_sed, 1446, 1448
forward, 547
forward_as_tuple, 559
forward_iterator_tag, 903
forward_list
before_begin, 807
cbefore_begin, 807
clear, 809
constructor, 807
emplace_after, 808
emplace_front, 807
erase_after, 808
front, 807
insert_after, 808
merge, 810
pop, 808
push_front, 808
remove, 810
remove_if, 810
resize, 809
reverse, 810
sort, 810
splice_after, 809
unique, 810
ForwardIterator, 894
ForwardRange, 944
FP_FAST_FMA, 1161
FP_FAST_FMAF, 1161
FP_FAST_FMAL, 1161

Index of library names

FP_ILOGB0, 1161
FP_ILOGBNAN, 1161
FP_INFINITY, 1161
FP_NAN, 1161
FP_NORMAL, 1161
FP_SUBNORMAL, 1161
FP_ZERO, 1161
fpclassify, 1161
fpos, 1312, 1316, 1323, 1324
  state, 1324
fpos_t, 1434
fprintf, 1434
fputc, 1434
fgets, 1434
fputs, 1434
fputwc, 756
fputws, 756
frac_digits
moneypunct, 1307
fread, 1434
free, 474, 618
freeze
  ostrstream, 1603
  stringstream, 1604
  stringstreambuf, 1599
freopen, 1434
frexp, 1161
frexpf, 1161
frexpl, 1161
from_address
  coroutine_handle, 513
from_bytes
  wstring_convert, 1610
from_chars, 713
from_chars_result, 711
  ec, 711
  ptr, 711
from_promise
  coroutine_handle, 513
from_stream
  day, 1218
  duration, 1201
  file_time, 1212
  gps_time, 1211
  local_time, 1213
  month, 1220
  month_day, 1228
  sys_time, 1206
  tai_time, 1209
  utc_time, 1208
  weekday, 1225
  year, 1223
  year_month, 1232
  year_month_day, 1235
from_sys
  utc_clock, 1207
from_time_t
  system_clock, 1205
from_utc
  gps_clock, 1210
  tai_clock, 1209
front
  basic_string, 731
  basic_string_view, 750
  forward_list, 807
  span, 876
  tzdb_list, 1251
front_insert_iterator, 911
  constructor, 912
  operator*, 912
  operator++, 912
  operator=, 912
front inserter, 912
fscanf, 1434
fseek, 1374, 1434
fsetpos, 1434
fstream, 1312, 1371
ftell, 1434
function, 675
  constructor, 676
  destructor, 677
  invocation, 677
  operator bool, 677
  operator!=, 678
  operator(), 677
  operator=, 677
  operator=, 678
  swap, 677, 678
  target, 677
  target_type, 677
function_name
  contract_violation, 498
future, 1536
  constructor, 1537
  get, 1537
  operator=, 1537
  share, 1537
  valid, 1538
  wait, 1538
  wait_for, 1538
  wait_until, 1538
future_category, 1532
future_errc, 1531
  make_error_code, 1532
  make_error_condition, 1532
future_error, 1533
  code, 1533
  constructor, 1533
  what, 1533
fwide, 756
fwprintf, 756
fwrite, 1434
fwscanf, 756
gamma_distribution, 1131
  alpha, 1132
  beta, 1132
  constructor, 1132
gbump
basic_streambuf, 1336
gcd, 1092
gcount
basic_istream, 1346
general
chars_format, 711
GENERALIZED_NONCOMMUTATIVE_SUM, 1084
GENERALIZED_SUM, 1084
generate, 1046
seed_seq, 1124
generate_canonical, 1125
generate_n, 1046
generic_category, 537, 538
generic_string
path, 1404
generic_u16string
path, 1404
generic_u32string
path, 1404
generic_u8string
path, 1404
generic_wstring
path, 1404
geometric_distribution, 1128
constructor, 1129
p, 1129
get, 877
array, 801
basic_istream, 1346, 1347
future, 1537
messages, 1308
money_get, 1303
num_get, 1289
pair, 552
reference_wrapper, 663
shared_future, 1540
shared_ptr, 631
subrange, 950
time_get, 1299
tuple, 561
unique_ptr, 622
variant, 583
get_allocator
basic_string, 738
basic_syncbuf, 1386
match_results, 1464
get_date
time_get, 1299
get_default_resource, 650
get_deleter
shared_ptr, 637
unique_ptr, 623
get_future
packaged_task, 1543
promise, 1535
get_id
this_thread, 1506
thread, 1506
get_if, 583, 584
get_info
time_zone, 1256
zoned_time, 1260
get_local_time
zoned_time, 1260
get_money, 1360
get_monthname
time_get, 1299
get_new_handler, 468, 495
get_pointer_safety, 606
get_sys_time
zoned_time, 1260
generate_n
temporary_buffer
zombie, 466
generate
terminate, 468, 500
generate_canonical
get_time
1361
time_get, 1299
generate_n
time_zone
zoned_time, 1260
generate
get_temporary_buffer
zombie, 466
generate
get_unexpected
zombie, 466
geometric_distribution
get_weekday
time_get, 1299
generate
get_wrapped
basic_syncbuf, 1386
generate
get_year
time_get, 1299
generate
getc
1434
generate
getchar
1434
generate
getenv
474, 515
generate
getline
basic_istream, 1347, 1348
basic_string, 743
generate
getloc
1451
basic_regex, 1455
basic_streambuf, 1334
ios_base, 1322
generate
gets
zombie, 466
generate
getwc
756
generate
getwchar
756
generate
getchar
global
locale, 1278
gmtime
1209
generate
good
basic_ios, 1328
generate
gps_clock, 1209
from_utc, 1210
now, 1210
to_utc, 1210
generate
gps_seconds, 1177
generate
gps_time, 1177
from_stream, 1211
operator<<, 1210
to_stream, 1211
generate
getptr
basic_streambuf, 1336
Index of library names
greater, 666, 668
operator(), 666
partial_ordering, 506
weak_ordering, 507
greater>, 666
operator(), 666
greater_equal, 667, 669
operator(), 667
greater_equal>, 667
operator(), 667

has_root_name
path, 1406

has_root_path
path, 1406

has_signaling_NaNaN numeric_limits, 483

has_stem
path, 1406

has_unique_object_representations, 695, 696

has_value
any, 590
optional, 571

has_virtual_destructor, 695

hash, 681
collate, 1297
coroutine_handle, 514
error_code, 542
monostate, 586
optional, 575

has_denorm_loss
numeric_limits, 483

has_extension
path, 1406

has_facet
locale, 1278

has_filename
path, 1406

has_infinity
numeric_limits, 483

has_parent_path
path, 1406

has_quiet_NaNaN numeric_limits, 483

has_relative_path
path, 1406

has_root_directory
path, 1406

hash_code

hasher
unordered associative containers, 784

hermite, 1174

hermitef, 1174

hermitel, 1174

hex, 1330

hex
chars_format, 711

hexfloat, 1330

high_resolution_clock, 1213

holds_alternative, 583

hours, 1177
time_of_day<sub-second duration>, 1248
time_of_day<hours>, 1246
time_of_day<minutes>, 1246
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time_of_day&lt;seconds&gt;</code></td>
<td>Represents the current time of day.</td>
<td>1247</td>
</tr>
<tr>
<td><code>HUGE_VAL</code></td>
<td>Represents a very large positive number.</td>
<td>1161</td>
</tr>
<tr>
<td><code>HUGE_VALF</code></td>
<td>Represents a very large positive floating-point number.</td>
<td>1161</td>
</tr>
<tr>
<td><code>HUGE_VALL</code></td>
<td>Represents a very large positive integer.</td>
<td>1161</td>
</tr>
<tr>
<td><code>hypot</code></td>
<td>Computes the Euclidean distance between two points.</td>
<td>1161</td>
</tr>
<tr>
<td><code>hypotf</code></td>
<td>Computes the Euclidean distance between two points for floating-point types.</td>
<td>1161</td>
</tr>
<tr>
<td><code>hypotl</code></td>
<td>Computes the Euclidean distance between two points for long integer types.</td>
<td>1161</td>
</tr>
<tr>
<td><code>id</code></td>
<td>Represents an identifier.</td>
<td>559</td>
</tr>
<tr>
<td><code>locale</code></td>
<td>Represents a locale.</td>
<td>1276</td>
</tr>
<tr>
<td><code>identity</code></td>
<td>Represents an identity.</td>
<td>671</td>
</tr>
<tr>
<td><code>ifstream</code></td>
<td>Represents an input file stream.</td>
<td>1312, 1371</td>
</tr>
<tr>
<td><code>ignore</code></td>
<td>Ignores input characters.</td>
<td>559</td>
</tr>
<tr>
<td><code>basic_istream</code></td>
<td>Represents an input stream.</td>
<td>1348</td>
</tr>
<tr>
<td><code>ilogb</code></td>
<td>Returns the integer part of the logarithm.</td>
<td>1161</td>
</tr>
<tr>
<td><code>ilogbf</code></td>
<td>Returns the integer part of the logarithm for floating-point types.</td>
<td>1161</td>
</tr>
<tr>
<td><code>ilogbl</code></td>
<td>Returns the integer part of the logarithm for long integer types.</td>
<td>1161</td>
</tr>
<tr>
<td><code>imag</code></td>
<td>Returns the imaginary part of a complex number.</td>
<td>1103</td>
</tr>
<tr>
<td><code>complex</code></td>
<td>Represents a complex number.</td>
<td>1099, 1101</td>
</tr>
<tr>
<td><code>imaxabs</code></td>
<td>Returns the absolute value of an integer.</td>
<td>1435</td>
</tr>
<tr>
<td><code>imaxdiv</code></td>
<td>Performs integer division.</td>
<td>1435</td>
</tr>
<tr>
<td><code>imaxdiv_t</code></td>
<td>Represents the type of the integer division result.</td>
<td>1435</td>
</tr>
<tr>
<td><code>imbue</code></td>
<td>Imbues a locale with a codecvt.</td>
<td>1451</td>
</tr>
<tr>
<td><code>codecvt</code></td>
<td>Represents a codecvt.</td>
<td>1285</td>
</tr>
<tr>
<td><code>in_avail</code></td>
<td>Returns the available input.</td>
<td>1334</td>
</tr>
<tr>
<td><code>basic_streambuf</code></td>
<td>Represents a basic streambuf.</td>
<td>1336</td>
</tr>
<tr>
<td><code>ios_base</code></td>
<td>Represents an ios base.</td>
<td>1322</td>
</tr>
<tr>
<td><code>in</code></td>
<td>Represents an input iterator.</td>
<td>1285</td>
</tr>
<tr>
<td><code>codecvt</code></td>
<td>Represents a codecvt.</td>
<td>1285</td>
</tr>
<tr>
<td><code>inclusive_scan</code></td>
<td>Performs an inclusive scan.</td>
<td>1089</td>
</tr>
<tr>
<td><code>increment</code></td>
<td>Represents an increment iterator.</td>
<td>1419</td>
</tr>
<tr>
<td><code>recursive_directory_iterator</code></td>
<td>Represents a recursive directory iterator.</td>
<td>1421</td>
</tr>
<tr>
<td><code>Incrementable</code></td>
<td>Represents an incrementable object.</td>
<td>893</td>
</tr>
<tr>
<td><code>incrementable_traits</code></td>
<td>Represents an incrementable traits.</td>
<td>886</td>
</tr>
<tr>
<td><code>independent_bits_engine</code></td>
<td>Represents an independent bits engine.</td>
<td>1119, 1120</td>
</tr>
<tr>
<td><code>index</code></td>
<td>Represents an index.</td>
<td>582</td>
</tr>
<tr>
<td><code>variant</code></td>
<td>Represents a variant.</td>
<td>582</td>
</tr>
<tr>
<td><code>weekday_indexed</code></td>
<td>Represents a weekday indexed sequence.</td>
<td>1226</td>
</tr>
<tr>
<td><code>year_month_weekday</code></td>
<td>Represents a year month weekday sequence.</td>
<td>1239</td>
</tr>
<tr>
<td><code>index_sequence</code></td>
<td>Represents an index sequence.</td>
<td>545</td>
</tr>
<tr>
<td><code>index_sequence_for</code></td>
<td>Represents an index sequence for.</td>
<td>545</td>
</tr>
<tr>
<td><code>indirect_array</code></td>
<td>Represents an indirect array.</td>
<td>1159</td>
</tr>
<tr>
<td><code>operator[]</code></td>
<td>Represents an array index operator.</td>
<td>1160</td>
</tr>
<tr>
<td><code>operator+=</code></td>
<td>Represents an increment assignment operator.</td>
<td>1160</td>
</tr>
<tr>
<td><code>operator-=</code></td>
<td>Represents a decrement assignment operator.</td>
<td>1160</td>
</tr>
<tr>
<td><code>operator&amp;</code></td>
<td>Represents a reference to an object.</td>
<td>1160</td>
</tr>
<tr>
<td><code>operator%=</code></td>
<td>Represents a modulus assignment operator.</td>
<td>1160</td>
</tr>
<tr>
<td><code>operator/=</code></td>
<td>Represents a division assignment operator.</td>
<td>1160</td>
</tr>
<tr>
<td><code>operator*=</code></td>
<td>Represents a multiplication assignment operator.</td>
<td>1160</td>
</tr>
<tr>
<td><code>value_type</code></td>
<td>Represents the value type of a type.</td>
<td>1159</td>
</tr>
<tr>
<td><code>INFINITY</code></td>
<td>Represents positive infinity.</td>
<td>1161</td>
</tr>
<tr>
<td><code>infinity</code></td>
<td>Represents positive infinity.</td>
<td>1161</td>
</tr>
<tr>
<td><code>numeric_limits</code></td>
<td>Represents a numeric limits container.</td>
<td>484</td>
</tr>
<tr>
<td><code>init</code></td>
<td>Represents an initialization list.</td>
<td>502</td>
</tr>
<tr>
<td><code>begin</code></td>
<td>Represents the beginning of an input range.</td>
<td>503</td>
</tr>
<tr>
<td><code>end</code></td>
<td>Represents the end of an input range.</td>
<td>503</td>
</tr>
<tr>
<td><code>size</code></td>
<td>Represents the size of an input range.</td>
<td>503</td>
</tr>
<tr>
<td><code>inner_allocators</code></td>
<td>Represents inner allocators.</td>
<td></td>
</tr>
<tr>
<td><code>scoped_allocator_adaptor</code></td>
<td>Represents a scoped allocator adaptor.</td>
<td>657</td>
</tr>
<tr>
<td><code>inner_allocator_type</code></td>
<td>Represents an inner allocator type.</td>
<td>656</td>
</tr>
<tr>
<td><code>inner_iterator</code></td>
<td>Represents an inner iterator.</td>
<td>981</td>
</tr>
<tr>
<td><code>split_view::inner_iterator</code></td>
<td>Represents a split view inner iterator.</td>
<td></td>
</tr>
<tr>
<td><code>inner_product</code></td>
<td>Performs an inner product.</td>
<td>1086</td>
</tr>
<tr>
<td><code>inplace_merge</code></td>
<td>Performs an inplace merge.</td>
<td>1065</td>
</tr>
<tr>
<td><code>InputIterator</code></td>
<td>Represents an input iterator.</td>
<td>894</td>
</tr>
<tr>
<td><code>InputRange</code></td>
<td>Represents an input range.</td>
<td>944</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>Represents an insert operation.</td>
<td></td>
</tr>
<tr>
<td><code>basic_string</code></td>
<td>Represents a basic string.</td>
<td>734, 735</td>
</tr>
<tr>
<td><code>deque</code></td>
<td>Represents a deque.</td>
<td>804</td>
</tr>
<tr>
<td><code>list</code></td>
<td>Represents a list.</td>
<td>814</td>
</tr>
<tr>
<td><code>map</code></td>
<td>Represents a map.</td>
<td>829</td>
</tr>
<tr>
<td><code>multimap</code></td>
<td>Represents a multimap.</td>
<td>834</td>
</tr>
<tr>
<td><code>unordered-associated containers</code></td>
<td>Represents unordered associated containers.</td>
<td>776</td>
</tr>
<tr>
<td><code>unordered-associated containers</code></td>
<td>Represents unordered associated containers.</td>
<td>788</td>
</tr>
<tr>
<td><code>unordered_map</code></td>
<td>Represents an unordered map.</td>
<td>848</td>
</tr>
<tr>
<td><code>unordered_multimap</code></td>
<td>Represents an unordered multimap.</td>
<td>854</td>
</tr>
<tr>
<td><code>vector</code></td>
<td>Represents a vector.</td>
<td>820</td>
</tr>
<tr>
<td><code>insert_after</code></td>
<td>Inserts after an iterator.</td>
<td></td>
</tr>
<tr>
<td><code>forward_list</code></td>
<td>Represents a forward list.</td>
<td>808</td>
</tr>
<tr>
<td><code>insert_iterator</code></td>
<td>Inserts after an iterator.</td>
<td>912</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>Represents a constructor.</td>
<td>913</td>
</tr>
<tr>
<td><code>operator*</code></td>
<td>Represents an unary operator.</td>
<td>913</td>
</tr>
<tr>
<td><code>operator++</code></td>
<td>Represents a unary increment operator.</td>
<td>913</td>
</tr>
<tr>
<td><code>operator-=</code></td>
<td>Represents a unary decrement assignment operator.</td>
<td>913</td>
</tr>
<tr>
<td><code>insert_or_assign</code></td>
<td>Inserts or assigns after an iterator.</td>
<td>830</td>
</tr>
<tr>
<td><code>map</code></td>
<td>Represents a map.</td>
<td>830</td>
</tr>
<tr>
<td><code>unordered_map</code></td>
<td>Represents an unordered map.</td>
<td>849</td>
</tr>
<tr>
<td><code>inserter</code></td>
<td>Represents an inserter.</td>
<td>913</td>
</tr>
<tr>
<td><code>int16_t</code></td>
<td>Represents a 16-bit integer.</td>
<td>487</td>
</tr>
<tr>
<td><code>int32_t</code></td>
<td>Represents a 32-bit integer.</td>
<td>487</td>
</tr>
</tbody>
</table>
int64_t, 487
int8_t, 487
int_fast16_t, 487
int_fast32_t, 487
int_fast64_t, 487
int_fast8_t, 487
int_least16_t, 487
int_least32_t, 487
int_least64_t, 487
int_least8_t, 487
INT_MAX, 486
INT_MIN, 486
int_type
  char_traits, 716
  wstring_convert, 1610
integer_sequence, 548
  value_type, 548
Integral, 523
integral_constant, 688
  value_type, 688
internal, 1330
intervals
  piecewise_constant_distribution, 1140
  piecewise_linear_distribution, 1142
intmax_t, 487
intptr_t, 487
invalid_argument, 530, 531, 593
  constructor, 531
Invocable, 528
INVOCES, 661, 662
invoke, 662
io_errc, 1315
  make_error_code, 1331
  make_error_condition, 1331
io_state
  zombie, 466
ios, 1312, 1316
ios_base, 1317
  constructor, 1323
  destructor, 1323
  failure, 1319
  flags, 1321
  fmtflags, 1319
  getloc, 1322
  imbue, 1322
  Init, 1321
  iostate, 1320
  iword, 1322
  openmode, 1320
  precision, 1321
  pword, 1323
  register_callback, 1323
  seekdir, 1320
  setf, 1321
  sync_with_stdio, 1322
  unsetf, 1321
  width, 1321, 1322
  xalloc, 1322
ios_base::failure, 1319
constructor, 1319
ios_base::Init, 1321
  constructor, 1321
  destructor, 1321
iostate
  ios_base, 1320
  iostream_category, 1331
iota, 1092
iota_view
  iota_view, 954
is
  ctype, 1280
  ctype<char>, 1283
is_absolute
  path, 1406
is_abstract, 690
is_aggregate, 691
is_always_equal
  allocator, 611
  allocator_traits, 610
  scoped_allocator_adaptor, 656
is_always_lock_free
  atomic, 1490
  atomic<floating-point>, 1490
  atomic<integral>, 1490
  atomic<T*>, 1490
  atomic_ref, 1483
  atomic_ref<floating-point>, 1483
  atomic_ref<integral>, 1483
  atomic_ref<T*>, 1483
is_arithmetic, 689
is_array, 689
is_assignable, 692
is_base_of, 697
is_bind_expression, 672
is_block_file, 1428
  directory_entry, 1416
is_bounded
  numeric_limits, 484
  is_bounded_array, 691
  is_character_file, 1428
  directory_entry, 1416
is_class, 689
is_clock, 1194
is_clock_v, 1177
is_compound, 690
is_const, 690
is_constructible, 691, 696
is_convertible, 697, 698
is_copy_assignable, 692
is_copy_constructible, 691
is_default_constructible, 691
is_destructible, 693
is_directory, 1428
  directory_entry, 1416
is_empty
  class, 690
  function, 1428
is_enum, 689
is_eq, 503
is_equal
  memory_resource, 646
is_error_code_enum, 535
is_error_condition_enum, 535
is_exact
  numeric_limits, 482
is_execution_policy, 710
is_fifo, 1429
directory_entry, 1416
is_final, 691
is_floating_point, 689
is_function, 689
is_fundamental, 690
is_geq, 503
is_gt, 503
is_heap, 1073, 1074
is_heap_until, 1074
is_iec559
  numeric_limits, 484
is_integer
  numeric_limits, 482
is_integral, 689
is_invocable, 697
is_invocable_r, 697
is_leap
  year, 1222
is_literal_type, 1604
  zombie, 466
is_literal_type_v
  zombie, 466
is_lock_free
  atomic, 1490
    atomic<floating-point>, 1490
    atomic<integral>, 1490
    atomic<T*>, 1490
  atomic_ref, 1484
    atomic_ref<floating-point>, 1484
    atomic_ref<integral>, 1484
    atomic_ref<T**>, 1484
is_lt, 503
is_lteq, 503
is_lvalue_reference, 689
is_member_function_pointer, 689
is_member_object_pointer, 689
is_member_pointer, 690
is_modulo
  numeric_limits, 484
is_move_assignable, 692
is_move_constructible, 691
is_neq, 503
is_nothrow_assignable, 695
is_nothrow_constructible, 694
is_nothrow_convertible, 697
is_nothrow_convertible_v, 682
is_nothrow_copy_assignable, 695
is_nothrow_copy_constructible, 694
is_nothrow_default_constructible, 694
is_nothrow_destructible, 695
is_nothrow_invocable, 698
is_nothrow_invocable_r, 698
is_nothrow_move_assignable, 695
is_nothrow_move_constructible, 695
is_nothrow_swappable, 695
is_nothrow_swappable_with, 695
is_null_pointer, 689
is_object, 690
is_open
  basic_filebuf, 1374
    basic_fstream, 1383
    basic_ifstream, 1379
    basic_ofstream, 1381
is_other, 1429
directory_entry, 1416
is_partitioned, 1062
is_permutation, 1036
is_placeholder, 672
is_pointer, 689
is_polymorphic, 690
is_reference, 689
is_regular_file, 1429
directory_entry, 1416
is_relative
  path, 1406
is_rvalue_reference, 689
is_scalar, 690
is_signed
  class, 691
    numeric_limits, 482
is_socket, 1429
directory_entry, 1416
is_sorted, 1058
is_sorted_until, 1058
is_standard_layout, 690
is_swappable, 693
is_swappable_with, 693
is_symlink, 1430
directory_entry, 1416
is_trivial, 690
is_trivially_assignable, 694
is_trivially_constructible, 693
is_trivially_copy_assignable, 694
is_trivially_copy_constructible, 694
is_trivially_copyable, 690
is_trivially_default_constructible, 694
is_trivially_destructible, 694
is_trivially_move_assignable, 694
is_trivially_move_constructible, 694
is_unbounded_array, 691
is_union, 689
is_unsigned, 691
is_void, 689
is_volatile, 690
isalnum, 755, 1278
isalpha, 755, 1278
iscntrl, 755, 1278
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>isctype</td>
<td>regex_traits, 1450</td>
</tr>
<tr>
<td></td>
<td>regular expression traits, 1475</td>
</tr>
<tr>
<td>isdigit</td>
<td>755, 1278</td>
</tr>
<tr>
<td>isfinite</td>
<td>1161</td>
</tr>
<tr>
<td>isgraph</td>
<td>755, 1278</td>
</tr>
<tr>
<td>isgreater</td>
<td>1161</td>
</tr>
<tr>
<td>isgreaterequal</td>
<td>1161</td>
</tr>
<tr>
<td>isinf</td>
<td>1161</td>
</tr>
<tr>
<td>issless</td>
<td>1161</td>
</tr>
<tr>
<td>islessequal</td>
<td>1161</td>
</tr>
<tr>
<td>islessgreater</td>
<td>1161</td>
</tr>
<tr>
<td>islower</td>
<td>755, 1278</td>
</tr>
<tr>
<td>isnan</td>
<td>1161</td>
</tr>
<tr>
<td>isnormal</td>
<td>1161</td>
</tr>
<tr>
<td>isprint</td>
<td>755, 1278</td>
</tr>
<tr>
<td>ispunct</td>
<td>755, 1278</td>
</tr>
<tr>
<td>ispace</td>
<td>755, 1278</td>
</tr>
<tr>
<td>istream</td>
<td>1312, 1339</td>
</tr>
<tr>
<td>istream_iterator</td>
<td>928</td>
</tr>
<tr>
<td></td>
<td>constructor, 929</td>
</tr>
<tr>
<td></td>
<td>destructor, 929</td>
</tr>
<tr>
<td></td>
<td>operator!=, 930</td>
</tr>
<tr>
<td></td>
<td>operator*, 929</td>
</tr>
<tr>
<td></td>
<td>operator++, 929</td>
</tr>
<tr>
<td></td>
<td>operator-&gt;, 929</td>
</tr>
<tr>
<td></td>
<td>operator==, 929, 930</td>
</tr>
<tr>
<td>istreambuf_iterator</td>
<td>931</td>
</tr>
<tr>
<td></td>
<td>constructor, 932</td>
</tr>
<tr>
<td></td>
<td>equal, 932</td>
</tr>
<tr>
<td></td>
<td>operator!=, 933</td>
</tr>
<tr>
<td></td>
<td>operator*, 932</td>
</tr>
<tr>
<td></td>
<td>operator++, 932</td>
</tr>
<tr>
<td></td>
<td>operator==, 932</td>
</tr>
<tr>
<td>istringstream</td>
<td>1312, 1362</td>
</tr>
<tr>
<td>istrstream</td>
<td>1601</td>
</tr>
<tr>
<td></td>
<td>constructor, 1601, 1602</td>
</tr>
<tr>
<td></td>
<td>rdbuf, 1602</td>
</tr>
<tr>
<td></td>
<td>str, 1602</td>
</tr>
<tr>
<td>isnunordered</td>
<td>1161</td>
</tr>
<tr>
<td>iswalnum</td>
<td>755, 1278</td>
</tr>
<tr>
<td>iswalpha</td>
<td>755</td>
</tr>
<tr>
<td>iswblank</td>
<td>755</td>
</tr>
<tr>
<td>iswctrl</td>
<td>755</td>
</tr>
<tr>
<td>iswctype</td>
<td>755</td>
</tr>
<tr>
<td>iswdigit</td>
<td>755</td>
</tr>
<tr>
<td>iswgraph</td>
<td>755</td>
</tr>
<tr>
<td>iswlower</td>
<td>755</td>
</tr>
<tr>
<td>iswprint</td>
<td>755</td>
</tr>
<tr>
<td>iswpunct</td>
<td>755</td>
</tr>
<tr>
<td>iswspace</td>
<td>755</td>
</tr>
<tr>
<td>iswupper</td>
<td>755</td>
</tr>
<tr>
<td>isxdigit</td>
<td>755, 1278</td>
</tr>
<tr>
<td>iter_difference_t</td>
<td>887</td>
</tr>
<tr>
<td>iter_move</td>
<td>890</td>
</tr>
<tr>
<td></td>
<td>common_iterator, 922</td>
</tr>
<tr>
<td></td>
<td>counted_iterator, 927</td>
</tr>
<tr>
<td></td>
<td>filter_view::iterator, 962</td>
</tr>
<tr>
<td></td>
<td>move_iterator, 917</td>
</tr>
<tr>
<td></td>
<td>reverse_iterator, 910</td>
</tr>
<tr>
<td>iter_swap</td>
<td>890, 1042</td>
</tr>
<tr>
<td></td>
<td>common_iterator, 922</td>
</tr>
<tr>
<td></td>
<td>counted_iterator, 927</td>
</tr>
<tr>
<td></td>
<td>filter_view::iterator, 962</td>
</tr>
<tr>
<td></td>
<td>join_view::iterator, 976</td>
</tr>
<tr>
<td></td>
<td>move_iterator, 917</td>
</tr>
<tr>
<td></td>
<td>reverse_iterator, 910</td>
</tr>
<tr>
<td></td>
<td>split_view::inner_iterator, 982</td>
</tr>
<tr>
<td></td>
<td>transform_view::iterator, 968</td>
</tr>
<tr>
<td>iter_value_t</td>
<td>887</td>
</tr>
<tr>
<td>Iterator</td>
<td>893</td>
</tr>
<tr>
<td>iterator</td>
<td>1604</td>
</tr>
<tr>
<td></td>
<td>basic_string, 723</td>
</tr>
<tr>
<td></td>
<td>basic_string_view, 747</td>
</tr>
<tr>
<td></td>
<td>filter_view, 961</td>
</tr>
<tr>
<td></td>
<td>filter_view::iterator, 961</td>
</tr>
<tr>
<td></td>
<td>iota_view::iterator, 955</td>
</tr>
<tr>
<td></td>
<td>join_view::iterator, 975</td>
</tr>
<tr>
<td></td>
<td>transform_view::iterator, 966</td>
</tr>
<tr>
<td>iterator_category</td>
<td>iterator_traits, 888</td>
</tr>
<tr>
<td></td>
<td>iterator_traits, 888</td>
</tr>
<tr>
<td></td>
<td>iterator_category, 888</td>
</tr>
<tr>
<td></td>
<td>pointer, 888</td>
</tr>
<tr>
<td></td>
<td>reference, 888</td>
</tr>
<tr>
<td>ivord</td>
<td>ios_base, 1322</td>
</tr>
<tr>
<td></td>
<td>jmp_buf, 516</td>
</tr>
<tr>
<td></td>
<td>join</td>
</tr>
<tr>
<td></td>
<td>thread, 1505</td>
</tr>
<tr>
<td></td>
<td>join_view</td>
</tr>
<tr>
<td></td>
<td>join_view, 973</td>
</tr>
<tr>
<td></td>
<td>joinable</td>
</tr>
<tr>
<td></td>
<td>thread, 1505</td>
</tr>
<tr>
<td>k</td>
<td>negative_binomial_distribution, 1129</td>
</tr>
<tr>
<td>key_comp</td>
<td>ordered associative containers, 775</td>
</tr>
<tr>
<td>key_compare</td>
<td>ordered associative containers, 774</td>
</tr>
<tr>
<td>key_compare</td>
<td>ordered associative containers, 774</td>
</tr>
<tr>
<td>key_eq</td>
<td>unordered associative containers, 787</td>
</tr>
<tr>
<td>key_equal</td>
<td>unordered associative containers, 784</td>
</tr>
<tr>
<td>key_type</td>
<td>ordered associative containers, 774</td>
</tr>
<tr>
<td></td>
<td>unordered associative containers, 784</td>
</tr>
<tr>
<td>kill_dependency</td>
<td>1482</td>
</tr>
<tr>
<td>knuth_b</td>
<td>1122</td>
</tr>
<tr>
<td>L_tmpnam</td>
<td>1434</td>
</tr>
<tr>
<td>labs</td>
<td>474</td>
</tr>
<tr>
<td>laguerre</td>
<td>1174</td>
</tr>
</tbody>
</table>
laguerref, 1174
laguerrel, 1174
lambda
exponential_distribution, 1131
last
span, 875
last_spec, 1216
last_write_time, 1430
directory_entry, 1417
latest
choose, 1177
launder, 495
LC_ALL, 1309
LC_COLLATE, 1309
LC_CTYPE, 1309
LC_MONETARY, 1309
LC_NUMERIC, 1309
LC_TIME, 1309
lcm, 1092
lconv, 1309
LDBL_DECIMAL_DIG, 487
LDBL_DIG, 487
LDBL_EPSILON, 487
LDBL_HAS_SUBNORM, 487
LDBL_MANT_DIG, 487
LDBL_MAX, 487
LDBL_MAX_10_EXP, 487
LDBL_MIN, 487
LDBL_MIN_10_EXP, 487
LDBL_MIN_EXP, 487
LDBL_TRUE_MIN, 487
ldexp, 1161
ldexpf, 1161
ldexpl, 1161
ldiv, 474
ldiv_t, 474
leap, 1261
date, 1262
operator!=, 1262
operator<, 1262
operator<=, 1262, 1263
operator==, 1262
operator>, 1262
operator>=, 1263
left, 1330
legendre, 1174
legendref, 1174
legendrel, 1174
length
basic_string, 730
basic_string_view, 750
codecvt, 1285
match_results, 1463
regex_traits, 1450
sub_match, 1456
length_error, 530, 531
constructor, 531
lerp, 1170
less, 667, 668
operator(), 667
partial_ordering, 506
strong_ordering, 508
weak_ordering, 507
less<, 667
operator(), 667
less_equal, 667, 669
operator(), 667
less_equal<, 667
operator(), 667
lexically_normal
path, 1406
lexically_proximate
path, 1407
lexically_relative
path, 1407
lexicographical_compare, 1078
lexicographical_compare_3way, 1079
lgamma, 1161
lgammaf, 1161
lgammal, 1161
line_number
contract_violation, 498
linear_congruential_engine, 1115
constructor, 1115, 1116
link, 1263
name, 1263
operator<, 1263
operator==, 1263
target, 1263
list, 811
constructor, 813
splice, 815
little
endian, 705
llabs, 474
lldiv, 474
lldiv_t, 474
LLONG_MAX, 486
LLONG_MIN, 486
llrint, 1161
llrintf, 1161
llrintl, 1161
llround, 1161
llroundf, 1161
llroundl, 1161
load
atomic, 1490
atomic<floating-point>, 1490
atomic<integral>, 1490
atomic<shared_ptr<T>>, 643
atomic<T*>, 1490
atomic<weak_ptr<T>>, 644
atomic_ref, 1484
atomic_ref<floating-point>, 1484
atomic_ref<integral>, 1484
atomic_ref<T*>, 1484
load_factor
unordered associative containers, 794
local_days, 1177
local_info, 1255
ambiguous, 1255
first, 1255
nonexistent, 1255
operator<<, 1255
result, 1255
second, 1255
unique, 1255
local_iterator
unordered associative containers, 785
local_seconds, 1177
local_t, 1177
local_time, 1177, 1213
from_stream, 1213
operator<<, 1213
to_stream, 1213
locale, 1451, 1455, 1474
category, 1274
classic, 1278
combine, 1277
destructor, 1277
facet, 1275
global, 1278
has_facet, 1278
id, 1276
name, 1277
operator!=, 1277
operator(), 1277
operator=, 1277
operator==, 1277
use_facet, 1278
localeconv, 1309
localtime, 1269
locate_zone, 1252
tzdb, 1250
zoned_traits<const time_zone*>, 1257
lock, 1524
shared_lock, 1522
unique_lock, 1518
weak_ptr, 640
lock_guard, 1515
constructor, 1515
destructor, 1516
log, 1161
complex, 1102
valarray, 1154
log10, 1161
complex, 1102
valarray, 1154
log10f, 1161
log10l, 1161
log1p, 1161
log1pf, 1161
log1pl, 1161
log2, 1161
log2f, 1161
Index of library names 1708
log2l, 1161
log2pl, 1104
logb, 1161
logbf, 1161
logbl, 1161
logf, 1161
logic_error, 530
constructor, 530, 531
logical_and, 669
operator(), 669
logical_and<>, 669
operator(), 669
logical_not, 670
operator(), 670
logical_not<>, 670
operator(), 670
logical_or, 669
operator(), 669
logical_or<>, 669
operator(), 669
log1l, 1161
lognormal_distribution, 1134
classic, 1135
construct, 1135
m, 1135
s, 1135
LONG_MAX, 486
LONG_MIN, 486
longjmp, 516
lookup_classname
regex_traits, 1450
regular expression traits, 1475
lookup_collatename
regex_traits, 1450
regular expression traits, 1475
lower_bound, 1060
ordered associative containers, 781
lowest	numeric_limits, 482
lrint, 1161
lrintf, 1161
lrintl, 1161
lround, 1161
lroundf, 1161
lroundl, 1161
m
fisher_f_distribution, 1137
lognormal_distribution, 1135
make12
time_of_day<sub-second duration>, 1248
time_of_day<hours>, 1246
time_of_day<minutes>, 1247
time_of_day<seconds>, 1247
make24
time_of_day<sub-second duration>, 1248
time_of_day<hours>, 1246
time_of_day<minutes>, 1246
time_of_day<seconds>, 1247
make_any, 590
make_error_code
  errc, 540
  future_errc, 1532
  io_errc, 1331
make_error_condition
  errc, 541
  future_errc, 1532
  io_errc, 1331
make_exception_ptr
  501
make_from_tuple
  560
make_heap
  1072
make_index_sequence
  545
make_integer_sequence
  548
make_move_iterator
  917
make_obj_using_allocator
  608
make_optional
  574, 575
make_pair
  551
make_preferred
  path, 1402
make_ready_at_thread_exit
  packaged_task, 1544
make_reverse_iterator
  910
make_shared
  632, 634, 635
make_signed
  699
make_tuple
  558
make_unique
  625
make_unsigned
  700
malloc
  474, 618, 1593
map
  826
clear, 781
constructor, 774, 829
contains, 781
count, 781
emplace, 776
emplace_hint, 776
equal_range, 781
erase, 780
extract, 779
find, 781
insert, 776, 829
insert_or_assign, 830
key_comp, 775
key_compare, 774
key_type, 774
lower_bound, 781
mapped_type, 774
merge, 780
node_type, 774
operator<, 829
operator==, 829
try_emplace, 830
upper_bound, 781
value_comp, 775
value_compare, 774
value_type, 774
mapped_type
  ordered associative containers, 774
  unordered associative containers, 784
mark_count
  basic_regex, 1455
  mask_array, 1158
  operator**, 1159
  operator+=, 1159
  operator-=, 1159
  operator/=, 1159
  operator<<=, 1159
  operator=, 1159
  operator>>=, 1159
  operator[]!, 1159
  operator%=-, 1159
  operator&=, 1159
  operator^=, 1159
  operator|, 1159
  value_type, 1158
match_any, 1446, 1447
match_continuous, 1446, 1447, 1471
match_default, 1446
match_flag_type, 1446, 1447, 1476
match_not_bol, 1446, 1447
match_not_bow, 1446, 1447
match_not_eol, 1446, 1447
match_not_eow, 1446, 1447
match_not_null, 1446, 1447, 1471
match_prev_avail, 1446, 1448, 1471
match_results, 1460, 1469, 1471
  begin, 1463
  constructor, 1462
  empty, 1463
  end, 1463
  format, 1464
  get_allocator, 1464
  length, 1463
  matched, 1461
  max_size, 1463
  operator!=, 1465
  operator=, 1462
  operator==, 1465
  operator[]!, 1463
  position, 1463
  prefix, 1463
  ready, 1463
  size, 1463
  str, 1463
  suffix, 1463
  swap, 1464, 1465
MATH_ERREXCEPT, 1161
math_errhandling, 1161
MATH_ERRNO, 1161
max
  1075
duration, 1197
duration_values, 1194
numeric_limits, 482
time_point, 1203
valarray, 1151
year, 1222
max_align_t
  473, 476
max_bucket_count
  unordered associative containers, 793
max_digits10
numeric_limits, 482
max_element, 1077
max_exponent
numeric_limits, 483
max_exponent10
numeric_limits, 483
max_length
codecvt, 1285
max_load_factor unordered associative containers, 794
max_size
allocator_traits, 611
array, 799
basic_string, 730
basic_string_view, 750
match_results, 1463
scoped_allocator_adaptor, 657
MB_CUR_MAX, 474
MB_LEN_MAX, 486
mblen, 474, 758
mbrlen, 756, 758
mbstowcs, 758
mbtowc, 756, 758
mbsinit, 756, 758
mbstate_t, 756
mbstowcs, 756, 758
mbstowcsa, 474, 758
mem_fn, 674
mem_fun
zombie, 466
mem_fun1_ref_t
zombie, 466
mem_fun1_t
zombie, 466
mem_fun_ref
zombie, 466
mem_fun_ref_t
zombie, 466
mem_fun_t
zombie, 466
memchr, 756
memcpy, 756
memcpy, 756
memmove, 756
memory_order, 1481
acq_rel, 1481
acquire, 1481
consume, 1481
relaxed, 1481
release, 1481
seq_cst, 1481
memory_order_acq_rel, 1481
memory_order_acquire, 1481
memory_order_consume, 1481
memory_order_relaxed, 1481
memory_order_release, 1481
memory_order_seq_cst, 1481
memory_resource, 646
allocate, 646
deallocate, 646
destructor, 646
do_allocate, 647
do_deallocate, 647
do_is_equal, 647
is_equal, 646
operator!=, 647
operator=, 646
operator==, 647
memset, 756
merge, 1064
forward_list, 810
list, 815
ordered associative containers, 780
unordered associative containers, 791
Mergeable, 903
mersenne_twister_engine, 1116
constructor, 1117
message
do_close, 1309
error_category, 538
error_code, 540
error_condition, 541
messages, 1308
close, 1308
do_get, 1309
do_open, 1308
get, 1308
open, 1308
messages_byname, 1309
microseconds, 1177
midpoint, 1093
milliseconds, 1177
min, 1074
duration, 1197
duration_values, 1193
numeric_limits, 481
time_point, 1203
valarray, 1151
year, 1222
min_element, 1076
min_exponent numeric_limits, 483
min_exponent10 numeric_limits, 483
minmax, 1075, 1076
minmax_element, 1077
minstd_rand, 1121
minstd_rand0, 1121
minus, 664
operator(), 664
minus<, 664
operator(), 664
minutes, 1177
time_of_day<sub-second duration>, 1248
time_of_day<minutes>, 1246
time_of_day<seconds>, 1247
mismatch, 1033
mktime, 1269
modf, 1161
modff, 1161
modfl, 1161
modulus, 665
operator(), 665
modulus<, 665
operator(), 665
money_get, 1303
do_get, 1303
get, 1303
money_put, 1304
do_put, 1305
put, 1305
moneypunct, 1305
curr_symbol, 1307
decimal_point, 1307
do_curr_symbol, 1307
do_decimal_point, 1307
do_frac_digits, 1307
do_grouping, 1307
do_neg_format, 1307
do_negative_sign, 1307
do_pos_format, 1307
do_positive_sign, 1307
do_thousands_sep, 1307rac_digits, 1307
grouping, 1307
negative_sign, 1307
positive_sign, 1307
thousands_sep, 1307
moneypunct_byname, 1307
monostate, 585
monotonic_buffer_resource, 653
constructor, 653, 654
destructor, 654
do_allocate, 654
do_deallocate, 654
do_is_equal, 654
release, 654
upstream_resource, 654
month, 1218
constructor, 1219
from_stream, 1220
month_day, 1227
month_day_last, 1228
month_weekday, 1229
month_weekday_last, 1230
ok, 1220
operator unsigned, 1219
operator+, 1220
operator++, 1219
operator+=, 1219
operator-, 1220
operator=, 1219
operator-=, 1220
operator<, 1220
operator<<, 1220
operator==, 1220
to_stream, 1220
year_month, 1231
year_month_day, 1234
year_month_day_last, 1236
year_month_weekday, 1239
year_month_weekday_last, 1241
month_day, 1227
creator, 1227
day, 1227
from_stream, 1228
month, 1227
ok, 1228
operator<, 1228
operator<<, 1228
operator==, 1228
to_stream, 1228
month_day_last, 1228
creator, 1228
month, 1229
ok, 1229
operator<, 1229
operator<<, 1229
operator==, 1229
year_month_day_last, 1236
month_weekday, 1229
creator, 1229
month, 1229
ok, 1229
operator<, 1230
operator==, 1230
weekday_indexed, 1229
month_weekday_last, 1230
creator, 1230
month, 1230
ok, 1230
operator<, 1231
operator==, 1230
weekday_last, 1230
months, 1177
Movable, 528
move
algorithm, 1040, 1041
basic_ios, 1327
function, 547
move_backward, 1041
move_if_noexcept, 548
move_iterator, 914
base, 915
creator, 915
iter_move, 917
iter_swap, 917
operator!=, 916
operator*, 915
operator+, 916, 917
operator++, 915
operator+=, 916
operator-, 916, 917
operator-=, 916
operator->, 1605
operator-=, 916
operator<, 916
operator<=, 916
operator+=, 916
operator->, 1605
operator-- , 916
operator<, 916
operator>=, 917
operator=, 915
operator==, 916
operator>, 916
operator<<, 1605
operator<<, 816
operator<<+, 915
operator<<+=, 916
operator<<-, 916
operator<> , 918
operator<<, 918
operator<<=, 918
MoveConstructible, 525
mt19937, 1121
mt19937_64, 1121
multimap, 831

clear, 781
constructor, 774, 834
contains, 781
count, 781
emplace, 776
emplace_hint, 776
equal_range, 781
erase, 780
extract, 779
find, 781
insert, 776
key_comp, 775
key_compare, 774
key_type, 774
lower_bound, 781
mapped_type, 774
merge, 780
node_type, 774
operator<, 840
operator<=, 840
upper_bound, 781
value_comp, 775
value_compare, 774
value_type, 774
mutex, 1509

shared_lock, 1523
unique_lock, 1520

n

chi_squared_distribution, 1135
fisher_f_distribution, 1137
name
error_category, 537, 538
link, 1263
locale, 1277
time_zone, 1256
type_index, 709
type_info, 497
NAN, 1161
nan, 1161
nanf, 1161
nanl, 1161
nanoseconds, 1177
narrow

basic_ios, 1327
ctype, 1280
ctype<char>, 1283
native
dian, 705
path, 1403
NDEBUG, 456
nearbyint, 1161
nearbyintf, 1161
nearbyintl, 1161
negate, 665
operator(), 665
negate<>, 665
operator(), 665
negation, 705

negative_binomial_distribution, 1129
constructor, 1129
k, 1129
p, 1130
negative_sign
moneypunct, 1307
nested_exception, 501

Index of library names
constructor, 502
nested_ptr, 502
rethrow_if_nested, 502
rethrow_nested, 502
throw_with_nested, 502
nested_ptr
nested_exception, 502
new
operator, 467, 468, 491–494, 618
new_delete_resource, 650
new_handler, 495
new_object
polymorphic_allocator, 649
next, 904, 906
subrange, 950
next_permutation, 1079
nextafter, 1161
nextafterf, 1161
nextafterl, 1161
nexttoward, 1161
nexttowardf, 1161
nexttowardl, 1161
noboolalpha, 1329
node_type
ordered associative containers, 774
unordered associative containers, 785
noemit_on_flush, 1358
none
bitset, 595
none_of, 1027
nonequal
strong_equality, 505
nonequivalent
strong_equality, 505
weak_equality, 504
nonexistent
local_info, 1255
nonexistent_local_time, 1252
constructor, 1252
noop_coroutine, 515
noop_coroutine_handle, 511
noop_coroutine_promise, 514
norm, 1103
complex, 1101
normal_distribution, 1133
constructor, 1134
mean, 1134
stddev, 1134
noshowbase, 1329
noshowpoint, 1329
noshowpos, 1329
noskipws, 1329
not1
zombie, 466
not2
zombie, 466
not_equal_to, 666, 668
operator(), 666
not_equal_to<>, 666
operator(), 666
not_fn, 671
nothrow, 490
nothrow_t, 490
notify_all
condition_variable, 1527
condition_variable_any, 1530
notify_all_at_thread_exit, 1525
notify_one
condition_variable, 1527
condition_variable_any, 1530
nounitbuf, 1330
nouppercase, 1329
now
file_clock, 1211
gps_clock, 1210
tai_clock, 1208
utc_clock, 1206
nth_element, 1059
NULL, 473–475, 756, 1269, 1309, 1434
null_memory_resource, 650
nullopt, 572
nullopt_t, 572
nullptr_t, 473, 475
num_get, 1288
do_get, 1289, 1291
get, 1289
num_put, 1292
do_put, 1292, 1294
put, 1292
numeric_limits, 479, 480
denorm_min, 484
digits, 482
digits10, 482
epsilon, 482
float_denorm_style, 483
has_denorm_loss, 483
has_infinity, 483
has_quiet_NaN, 483
has_signaling_NaN, 483
infinity, 484
is_bounded, 484
is_exact, 482
is_iec559, 484
is_integer, 482
is_modulo, 484
is_signed, 482
lowest, 482
max, 482
max_digits10, 482
max_exponent, 483
max_exponent10, 483
min, 481
min_exponent, 483
min_exponent10, 483
quiet_NaN, 484
radix, 482
round_error, 483
round_style, 485
Index of library names
Index of library names

atomic_ref, 1484
operator basic_string
sub_match, 1456
operator basic_string_view
basic_string, 738
operator bool
basic_ios, 1328
basic_istream::sentry, 1343
basic_ostream::sentry, 1353
coroutine_handle, 513
coroutine_handle<noop_coroutine_-
promise>, 514
error_code, 540
error_condition, 541
function, 677
optional, 571
shared_lock, 1523
shared_ptr, 632
unique_lock, 1520
unique_ptr, 623
operator const filesystem::path&
directory_entry, 1415
operator int
year, 1222
operator local_days
year_month_day, 1234
year_month_day_last, 1237
year_month_weekday, 1239
year_month_weekday_last, 1241
operator local_time
zoned_time, 1260
operator partial_ordering
strong_ordering, 509
weak_ordering, 508
operator precision
time_of_day<sub-second duration>, 1248
time_of_day<hours>, 1246
time_of_day<minutes>, 1246
time_of_day<seconds>, 1247
operator shared_ptr<T>
atomic<shared_ptr<T>>, 643
operator string_type
path, 1403
operator strong_equality
strong_ordering, 509
operator sys_days
year_month_day, 1234
year_month_day_last, 1237
year_month_weekday, 1239
year_month_weekday_last, 1241
operator sys_time
zoned_time, 1260
operator T*
atomic<T*>, 1490
atomic_ref<T*>, 1484
operator T&
reference_wrapper, 663
operator unsigned

signaling.NaN, 484
tinyness_before, 484
traps, 484
numeric_limits<bool>, 485
numpunct, 1295
decimal_point, 1295
do_decimal_point, 1296
do_falsename, 1296
do_grouping, 1296
do_thousands_sep, 1296
do_truename, 1296
falsename, 1296
grouping, 1296
thousands_sep, 1295
truename, 1296
numpunct_byname, 1296
oct, 1330
offset
sys_info, 1254
offsetof, 473, 476, 1593
ofstream, 1312, 1371
ok
day, 1218
month, 1220
month_day, 1228
month_day_last, 1229
month_weekday, 1229
month_weekday_last, 1230
weekday, 1224
weekday_indexed, 1226
weekday_last, 1227
year, 1222
year_month, 1232
year_month_day, 1234
year_month_day_last, 1237
year_month_weekday, 1239
year_month_weekday_last, 1241
once_flag, 1524
open
basic_filebuf, 1374
basic_fstream, 1383, 1384
basic_ifstream, 1379
basic_ofstream, 1381
messages, 1308
open_mode
zombie, 466
openmode
ios_base, 1320
operator floating-point
atomic<floating-point>, 1490
atomic_ref<floating-point>, 1484
operator integral
atomic<integral>, 1490
atomic_ref<integral>, 1484
operator PairLike
 subrange, 949
operator type
atomic, 1490
Index of library names
equal_to, 666
equal_to<> , 666
function, 677
greater, 666
greater< , 666
greater_equal, 667
greater_equal<> , 667
less, 667
less<> , 667
less_equal, 667
less_equal<> , 667
locale, 1277
logical_and, 669
logical_and<> , 669
logical_not, 670
logical_not<> , 670
logical_or, 669
logical_or<> , 669
minus, 664
minus<> , 664
modulus, 665
modulus<> , 665
multiplies, 664
multiplies<> , 664
negate, 665
negate<> , 665
not_equal_to, 666
not_equal_to<> , 666
owner_less, 640
packaged_task, 1544
plus, 664
plus<> , 664
random_device, 1123
reference_wrapper, 663
operator*
back_insert_iterator, 911
common_iterator, 920
complex, 1100
counted_iterator, 925
duration, 1197
filter_view::iterator, 962
front_insert_iterator, 912
insert_iterator, 913
iota_view::iterator, 955
istream_iterator, 929
istreambuf_iterator, 932
move_iterator, 915
optional, 571
ostream_iterator, 931
ostreambuf_iterator, 933
regex_iterator, 1471
regex_token_iterator, 1474
reverse_iterator, 908
shared_ptr, 632
split_view::outer_iterator, 979
unique_ptr, 622
tvalarray, 1152
operator==
complex, 1099, 1100
duration, 1197
gslice_array, 1158
indirect_array, 1160
mask_array, 1159
slice_array, 1155
tvalarray, 1150
operator+
basic_string, 740, 741
complex, 1100
counted_iterator, 925
day, 1218
duration, 1196, 1203
iota_view::iterator, 956, 957
month, 1220
move_iterator, 916, 917
reverse_iterator, 908, 910
time_point, 1203
transform_view::iterator, 968
tvalarray, 1150, 1152
weekday, 1225
year, 1222
year_month, 1232
year_month_day, 1235
year_month_day_last, 1237
year_month_weekday, 1239
year_month_weekday_last, 1241
operator++
atomic<integral>, 1497
atomic<T*>, 1497
atomic_ref<integral>, 1488
atomic_ref<T*>, 1488
back_insert_iterator, 911
common_iterator, 921
counted_iterator, 925
day, 1217
directory_iterator, 1419
duration, 1196
filter_view::iterator, 962
front_insert_iterator, 912
insert_iterator, 913
iota_view::iterator, 955
istream_iterator, 929
istreambuf_iterator, 932
join_view::iterator, 975
month, 1219
move_iterator, 915
ostream_iterator, 931
ostreambuf_iterator, 934
recursive_directory_iterator, 1421
regex_iterator, 1471
regex_token_iterator, 1474
reverse_iterator, 908
split_view::inner_iterator, 981
split_view::outer_iterator, 979
time_point, 1202
transform_view::iterator, 966, 967
weekday, 1224
year, 1221
operator+=
Index of library names
Index of library names

.. index::

   gslice_array, 1158
   indirect_array, 1160
   mask_array, 1159
   path, 1401, 1402
   slice_array, 1155
   valarray, 1150

operator<
   basic_string_view, 754
   coroutine_handle, 514
   counted_iterator, 927
   day, 1218
   directory_entry, 1417
   duration, 1198
   error_category, 538
   error_code, 542
   error_condition, 542
   iota_view::iterator, 956
   leap, 1262
   link, 1263
   monostate, 585
   month, 1220
   month_day, 1228
   month_day_last, 1229
   move_iterator, 916
   optional, 572–574
   pair, 551
   partial_ordering, 507
   path, 1408
   queue, 867
   reverse_iterator, 909
   shared_ptr, 636
   stack, 871
   strong_ordering, 509
   sub_match, 1457–1460
   sys_time, 1262
   thread::id, 1504
   time_point, 1204
   time_zone, 1256
   transform_view::iterator, 967
   tuple, 562
   type_index, 709
   unique_ptr, 626
   valarray, 1153
   variant, 584
   weak_ordering, 508
   year, 1222
   year_month, 1232
   year_month_day, 1234
   year_month_day_last, 1237

operator<<
   basic_ostream, 1354–1356, 1358
   basic_string, 743
   basic_string_view, 754
   bitset, 595, 596
   byte, 476
   complex, 1101
   day, 1218
   duration, 1200
   error_code, 540
file_time, 1212
gps_time, 1210
local_info, 1255
local_time, 1213
month, 1220
month_day, 1228
month_day_last, 1229
month_weekday, 1230
month_weekday_last, 1231
path, 1408
shared_ptr, 638
sub_match, 1460
sys_days, 1205
sys_info, 1254
sys_time, 1205
tai_time, 1209
thread::id, 1504
time_of_day, 1250
  <sub-second duration>, 1250
time_of_day, 1248
  <hours>, 1248
time_of_day, 1249
  <minutes>, 1249
time_of_day, 1249
  <seconds>, 1249
unique_ptr, 627
utc_time, 1207
valarray, 1152
weekday, 1225
weekday_indexed, 1226
weekday_last, 1227
year, 1222
year_month, 1232
year_month_day, 1235
year_month_day_last, 1237
year_month_weekday, 1240
year_month_weekday_last, 1242
zoned_time, 1261

operator<<=
   bitset, 594
   byte, 476
   gslice_array, 1158
   indirect_array, 1160
   mask_array, 1159
   slice_array, 1155
   valarray, 1150

operator<=, 453, 1596
   basic_string_view, 754
   coroutine_handle, 514
   counted_iterator, 927
   directory_entry, 1417
   duration, 1198
   iota_view::iterator, 956
   leap, 1262, 1263
   monostate, 585
   move_iterator, 917
   optional, 573, 574
   pair, 551
   partial_ordering, 507
   path, 1409
   queue, 867
   reverse_iterator, 909
   shared_ptr, 636
stack, 871
strong_ordering, 509
sub_match, 1457–1460
sys_time, 1262, 1263
thread::id, 1504
time_point, 1204
transform_view::iterator, 967
tuple, 562
type_index, 709
unique_ptr, 626, 627
valarray, 1153
variant, 584
weak_ordering, 508

operator<=>
partial_ordering, 507
strong_equality, 506
weak_ordering, 508

operator=
allocator, 611
any, 588, 589
atomic, 1490
atomic<floating-point>, 1490
atomic<integral>, 1490
atomic<shared_ptr<T>>, 643
atomic<T*>, 1490
atomic<weak_ptr<T>>, 644
atomic_ref, 1484
atomic_ref<floating-point>, 1484
atomic_ref<integral>, 1484
atomic_ref<T*>, 1484
back_insert_iterator, 911
basic_filebuf, 1374
basic_ifstream, 1379
basic_iostream, 1351
basic_istream, 1342
basic_stringstream, 1368
basic_ofstream, 1381
basic_ostream, 1353
basic_ostringstream, 1369
basic_stringstream, 1388
basic_stringview, 1454
basic_stringbuf, 1335
basic_string, 729, 730
basic_stringbuf, 1365
basic_stringstream, 1371
basic_sstream, 1385
common_iterator, 920
coroutine_handle, 513
counted_iterator, 924
directory_iterator, 1419
enable_shared_from_this, 641
error_code, 539
error_condition, 541
exception, 499
front_insert_iterator, 912
function, 677
future, 1537
gslice_array, 1158
indirect_array, 1160
insert_iterator, 913
locale, 1277
mask_array, 1159
match_results, 1462
memory_resource, 646
move_iterator, 915
move_sentinel, 918
optional, 567–569
oststream_iterator, 930
oststreambuf_iterator, 933
packaged_task, 1543
pair, 550, 551
path, 1401
promise, 1535
recursive_directory_iterator, 1421
reference_wrapper, 663
reverse_iterator, 908
shared_future, 1539, 1540
shared_lock, 1522
shared_ptr, 631
slice_array, 1155, 1156
span, 875
thread, 1505
tuple, 557, 558
unique_lock, 1518
unique_ptr, 622, 625
valarray, 1147, 1148
variant, 580
weak_ptr, 639
zoned_time, 1260

operator==
allocator, 611
any, 588, 589
atomic, 1490
atomic_string_view, 754
bitset, 595
coroutine_iterator, 921
complex, 1100
coroutine_handle, 514
counted_iterator, 926
day, 1218
directory_entry, 1417
duration, 1198
error_category, 538
error_code, 541, 542
error_condition, 542
filter_view::iterator, 962
filter_view::sentinel, 963
function, 678
iota_view::iterator, 956
iota_view::sentinel, 957
istream_iterator, 929, 930
istreambuf_iterator, 932
join_view::iterator, 976
join_view::sentinel, 976
leap, 1262
link, 1263
locale, 1277
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>match_results</td>
<td>1465</td>
</tr>
<tr>
<td>memory_resource</td>
<td>647</td>
</tr>
<tr>
<td>monostate</td>
<td>585</td>
</tr>
<tr>
<td>month</td>
<td>1220</td>
</tr>
<tr>
<td>month_day</td>
<td>1228</td>
</tr>
<tr>
<td>month_day_last</td>
<td>1229</td>
</tr>
<tr>
<td>month_weekday</td>
<td>1230</td>
</tr>
<tr>
<td>month_weekday_last</td>
<td>1230</td>
</tr>
<tr>
<td>move_iterator</td>
<td>916</td>
</tr>
<tr>
<td>optional</td>
<td>572–574</td>
</tr>
<tr>
<td>pair</td>
<td>551</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>507</td>
</tr>
<tr>
<td>path</td>
<td>1408</td>
</tr>
<tr>
<td>polymorphic_allocator</td>
<td>649</td>
</tr>
<tr>
<td>queue</td>
<td>867</td>
</tr>
<tr>
<td>regex_iterator</td>
<td>1470</td>
</tr>
<tr>
<td>regex_token_iterator</td>
<td>1472, 1474</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>909</td>
</tr>
<tr>
<td>scoped_allocator_adaptor</td>
<td>658</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>635, 636</td>
</tr>
<tr>
<td>split_view::inner_iterator</td>
<td>981, 982</td>
</tr>
<tr>
<td>split_view::outer_iterator</td>
<td>980</td>
</tr>
<tr>
<td>stack</td>
<td>871</td>
</tr>
<tr>
<td>strong_equality</td>
<td>505</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>509</td>
</tr>
<tr>
<td>sub_match</td>
<td>1457–1460</td>
</tr>
<tr>
<td>sys_time</td>
<td>1262</td>
</tr>
<tr>
<td>take_view::sentinel</td>
<td>971</td>
</tr>
<tr>
<td>thread::id</td>
<td>1504</td>
</tr>
<tr>
<td>time_point</td>
<td>1203</td>
</tr>
<tr>
<td>time_zone</td>
<td>1256</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>967</td>
</tr>
<tr>
<td>transform_view::sentinel</td>
<td>969</td>
</tr>
<tr>
<td>tuple</td>
<td>561</td>
</tr>
<tr>
<td>type_index</td>
<td>708</td>
</tr>
<tr>
<td>type_info</td>
<td>497</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>626</td>
</tr>
<tr>
<td>unreachable_sentinel_t</td>
<td>928</td>
</tr>
<tr>
<td>valarray</td>
<td>1153</td>
</tr>
<tr>
<td>variant</td>
<td>584</td>
</tr>
<tr>
<td>weak_equality</td>
<td>505</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>508</td>
</tr>
<tr>
<td>weekday</td>
<td>1225</td>
</tr>
<tr>
<td>weekday_indexed</td>
<td>1226</td>
</tr>
<tr>
<td>weekday_last</td>
<td>1227</td>
</tr>
<tr>
<td>year</td>
<td>1222</td>
</tr>
<tr>
<td>year_month</td>
<td>1232</td>
</tr>
<tr>
<td>year_month_day</td>
<td>1234</td>
</tr>
<tr>
<td>year_month_day_last</td>
<td>1237</td>
</tr>
<tr>
<td>year_month_weekday</td>
<td>1239</td>
</tr>
<tr>
<td>year_month_weekday_last</td>
<td>1241</td>
</tr>
<tr>
<td>zoned_time</td>
<td>1260</td>
</tr>
<tr>
<td>operator&gt;</td>
<td>452, 1596</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>754</td>
</tr>
<tr>
<td>coroutine_handle</td>
<td>514</td>
</tr>
<tr>
<td>counted_iterator</td>
<td>927</td>
</tr>
<tr>
<td>directory_entry</td>
<td>1417</td>
</tr>
<tr>
<td>duration</td>
<td>1198</td>
</tr>
<tr>
<td>iota_view::iterator</td>
<td>956</td>
</tr>
<tr>
<td>leap</td>
<td>1262</td>
</tr>
<tr>
<td>monostate</td>
<td>585</td>
</tr>
<tr>
<td>move_iterator</td>
<td>916</td>
</tr>
<tr>
<td>optional</td>
<td>573, 574</td>
</tr>
<tr>
<td>pair</td>
<td>551</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>507</td>
</tr>
<tr>
<td>path</td>
<td>1409</td>
</tr>
<tr>
<td>queue</td>
<td>867</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>909</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>636</td>
</tr>
<tr>
<td>stack</td>
<td>871</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>509</td>
</tr>
<tr>
<td>sub_match</td>
<td>1457–1460</td>
</tr>
<tr>
<td>sys_time</td>
<td>1262</td>
</tr>
<tr>
<td>thread::id</td>
<td>1504</td>
</tr>
<tr>
<td>time_point</td>
<td>1204</td>
</tr>
<tr>
<td>transform_view::iterator</td>
<td>967</td>
</tr>
<tr>
<td>tuple</td>
<td>562</td>
</tr>
<tr>
<td>type_index</td>
<td>709</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>626, 627</td>
</tr>
<tr>
<td>valarray</td>
<td>1153</td>
</tr>
<tr>
<td>variant</td>
<td>584</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>508</td>
</tr>
<tr>
<td>operator&gt;&gt;</td>
<td>453, 1596</td>
</tr>
<tr>
<td>basic_istream</td>
<td>1343–1345, 1350</td>
</tr>
<tr>
<td>basic_string</td>
<td>742</td>
</tr>
<tr>
<td>bitset</td>
<td>595, 596</td>
</tr>
<tr>
<td>byte</td>
<td>476</td>
</tr>
<tr>
<td>complex</td>
<td>1100</td>
</tr>
</tbody>
</table>
Index of library names

N4810

atomic<integral>, 1494
atomic_ref<integral>, 1486
bitset, 594
byte, 477
gslice_array, 1158
indirect_array, 1160
mask_array, 1159
slice_array, 1155
valarray, 1150
operator~
  bitset, 594
  byte, 477
  valarray, 1150
operator--
  counted_iterator, 925
operator|=
  atomic<integral>, 1494
  atomic_ref<integral>, 1486
bitset, 594
byte, 477
gslice_array, 1158
indirect_array, 1160
mask_array, 1159
slice_array, 1155
valarray, 1150
operator||
  valarray, 1153
optional, 564
  constructor, 565, 566
destructor, 567
emplace, 570
has_value, 571
operator bool, 571
operator!=, 572–574
operator*, 571
operator->, 571
operator<, 572–574
operator<=, 571
operator==, 572–574
operator>, 573, 574
operator>=, 573, 574
reset, 572
swap, 570
value, 571
value_or, 571, 572
value_type, 564
options
  recursive_directory_iterator, 1421
synchronized_pool_resource, 652
unsynchronized_pool_resource, 652
ostream, 1312, 1340
ostream_iterator, 930
counter, 930
operator*, 931
operator++, 931
operator=, 930
ostreambuf_iterator, 933
    constructor, 933
    failed, 934
    operator*, 933
    operator++, 934
    operator=, 933
osstringstream, 1312, 1362
ostreamstream, 1602
    constructor, 1602
    freeze, 1603
    pcount, 1603
    rdbuf, 1603
    str, 1603
osyncstream, 1312, 1384
out
codecvt, 1285
out_of_range, 530, 531, 593–595
    constructor, 532
outer_allocator
    scoped_allocator_adaptor, 657
outer_allocator_type
    scoped_allocator_adaptor, 655
outer_iterator
    split_view::outer_iterator, 979
output_iterator_tag, 903
OutputIterator, 894
OutputRange, 944
overflow
    basic_filebuf, 1376
    basic_streambuf, 1338
    basic_stringbuf, 1366
    strstreambuf, 1599
overflow_error, 530, 532, 593, 595
    constructor, 532, 533
owner_before
    shared_ptr, 632
    weak_ptr, 640
owner_less, 640
operator(), 640
owns_lock
    shared_lock, 1523
    unique_lock, 1520

P
 bernoulli_distribution, 1127
 binomial_distribution, 1128
 geometric_distribution, 1129
 negative_binomial_distribution, 1130
packaged_task, 1542
    constructor, 1543
destructor, 1543
get_future, 1543
make_ready_at_thread_exit, 1544
operator(), 1544
operator=, 1543
reset, 1544
swap, 1543, 1544
valid, 1543
pair, 548, 549, 557, 558
    constructor, 549, 550
get, 552
operator!=, 551
operator<, 551
operator<=, 551
operator>, 551
operator>=, 551
swap, 551
par, 711
par_unseq, 711
param
    seed_seq, 1124
parent_path
    path, 1405
parse, 1266–1269
partial_order, 510
partial_ordering, 506
equivalent, 506
greater, 506
less, 506
operator weak_equality, 507
operator!=, 507
operator<, 507
operator<=, 507
operator>, 507
operator>=, 507
unordered, 506
partial_sort, 1056
partial_sort_copy, 1057
partial_sum, 1087
partition, 1062
partition_copy, 1063
partition_point, 1064
path, 1394
    append, 1402
    assign, 1401
    begin, 1408
c_str, 1403
clear, 1402
compare, 1404, 1405
concat, 1402
destructor, 1400
directory_entry, 1415
eempty, 1406
deend, 1408
directory, 1405
dirname, 1405
generic_string, 1404
generic_u16string, 1404
generic_u32string, 1404
generic_u8string, 1404
generic_wstring, 1404
has_extension, 1406
new_object, 649
operator!=, 649
operator==, 649
resource, 649
select_on_container_copy_construction, 649
value_type, 647
pool_options, 650
largest_required_pool_block, 652
max_blocks_per_chunk, 651
pop
forward_list, 808
priority_queue, 869
recursive_directory_iterator, 1422
pop_back
basic_string, 736
pop_heap, 1072
position
match_results, 1463
positive_sign
moneypunct, 1307
pow, 1103, 1161
complex, 1102
valarray, 1154
powf, 1161
powl, 1161
pptr
basic_streambuf, 1336
precision
ios_base, 1279, 1321
Predicate, 529
preferred_separator
path, 1397
prefix
match_results, 1463
prev, 904, 906
subrange, 950
prev_permutation, 1080
PRIuFASTN, 1435
PRIuLEASTN, 1435
PRIuMAX, 1435
PRIuN, 1435
PRIuPTR, 1435
PRIxFASTN, 1435
PRIxLEASTN, 1435
PRIxMAX, 1435
PRIxN, 1435
PRIxPTR, 1435
probabilities
discrete_distribution, 1139
proj
complex, 1101
projected, 901
promise, 1534
coroutine_handle, 514
coroutine_handle<noop_coroutine_-
promise>, 515
destructor, 1535
get_future, 1535
operator=, 1535
set_exception, 1535
set_exception_at_thread_exit, 1536
set_value, 1535
set_value_at_thread_exit, 1536
swap, 1535, 1536
propagate_on_container_copy_assignment
allocator_traits, 610
scoped_allocator_adaptor, 656
propagate_on_container_move_assignment
allocator, 611
allocated_traits, 610
scoped_allocator_adaptor, 656
propagate_on_container_swap
allocated_traits, 610
scoped_allocator_adaptor, 656
proximate, 1430
proxy
istreambuf_iterator, 932
ptr
from_chars_result, 711
to_chars_result, 711
ptr_fun
zombie, 466
ptrdiff_t, 473
pubimbue
basic_streambuf, 1334
pubseekoff
basic_streambuf, 1334
pubseekpos
basic_streambuf, 1334
pubseekbuf

Index of library names
basic_streambuf, 1334
pubsync
basic_streambuf, 1334
push
priority_queue, 869
push_back
basic_string, 733
deque, 804
push_front
deque, 804
forward_list, 808
push_heap, 1071
put
basic_ostream, 1357
money_put, 1305
num_put, 1292
time_put, 1302
put_money, 1360
put_time, 1361
putback
basic_istream, 1348
putc, 1434
putchar, 1434
putenv, 515
puts, 1434
putwc, 756
putwchar, 756
pword
ios_base, 1323
qsort, 474, 1093
queue, 865
swap, 867
quick_exit, 455, 474, 489
quiet_NaN
numeric_limits, 484
quoted, 1361, 1362
radix
numeric_limits, 482
raise, 516
rand, 474, 1142
discouraged, 1142
RAND_MAX, 474
random_access_iterator_tag, 903
random_device, 1122
constructor, 1122
entropy, 1123
operator(), 1123
random_shuffle
zombie, 466
RandomAccessIterator, 895
RandomAccessRange, 944
Range, 942
range_error, 530, 532
constructor, 532
<ranges>, 936
ranges::swap, 523
rank, 696
ranlux24, 1122
ranlux24_base, 1122
ranlux48, 1122
ranlux48_base, 1122
ratio, 705, 706
ratio_equal, 707
ratio_greater, 708
ratio_greater_equal, 708
ratio_less, 707
ratio_less_equal, 708
ratio_not_equal, 707
raw_storage_iterator
zombie, 466
rbegin
basic_string, 730
basic_string_view, 749
span, 876
rbegin(C&), 934
rbegin(initializer_list<E>), 934
rbegin(T (&array)[N]), 934
rdbuf
basic_fstream, 1383
basic_ifstream, 1379
basic_ios, 1326
basic_istream, 1368
basic_ofstream, 1381
basic_ostringstream, 1370
basic_stringstream, 1371
istream, 1602
ostringstream, 1603
strstream, 1604
wbuffer_convert, 1612
rdstate
basic_ios, 1328
read
basic_istream, 1348
read_symlink, 1431
Readable, 891
readable_traits, 887
readsome
basic_istream, 1348
ready
match_results, 1463
real, 1103
complex, 1099, 1101
realloc, 474, 618, 1593
rebind
pointer_traits, 604
rebind_allocator
allocator_traits, 610
recursion_pending
recursive_directory_iterator, 1421
recursive_directory_iterator, 1419
constructor, 1420, 1421
depth, 1421
disable_recursion_pending, 1422
increment, 1421
operator++, 1421
operator=, 1421
Index of library names
1725
options, 1421
pop, 1422
recursion_pending, 1421
recursive_mutex, 1509
recursive_timed_mutex, 1511
reduce, 1085
ref
reference_wrapper, 663
ref_view, 959
reference
basic_string, 723
basic_string_view, 747
iterator_traits, 888
reference_wrapper, 662
constructor, 662, 663
cref, 663
get, 663
operator Tk, 663
operator(), 663
operator=, 663
ref, 663
refresh
directory_entry, 1415
regex, 1440
regex_constants, 1446
error_type, 1448, 1449
match_flag_type, 1446
syntax_option_type, 1446
regex_error, 1449, 1451, 1475, 1476
constructor, 1449
regex_iterator, 1469
constructor, 1470
increment, 1471
operator!=, 1470
operator*, 1471
operator++, 1471
operator->, 1471
operator==, 1470
regex_match, 1465, 1466
regex_replace, 1468, 1469
regex_search, 1467, 1468
regex_token_iterator, 1471
constructor, 1473
end-of-sequence, 1472
operator!=, 1474
operator*, 1474
operator++, 1474
operator->, 1474
operator==, 1472, 1474
regex_traits, 1449
char_class_type, 1449
isctype, 1450
length, 1450
lookup_classname, 1450
lookup_collatename, 1450
transform, 1450
transform_primary, 1450
translate, 1450
translate_nocase, 1450
value, 1451
register_callback
ios_base, 1323
Regular, 528
regular expression traits
isctype, 1475
lookup_classname, 1475
lookup_collatename, 1475
transform_primary, 1475
RegularInvocable, 529
rehash
unordered associative containers, 794
reinterpret_pointer_cast
shared_ptr, 637
rel_ops, 544
Relation, 529
relative, 1431
relative_path
path, 1405
relaxed
memory_order, 1481
release
memory_order, 1481
monotonic_buffer_resource, 654
shared_lock, 1523
synchronized_pool_resource, 652
unique_lock, 1520
unique_ptr, 623
unsynchronized_pool_resource, 652
reload_tzdb, 1252
remainder, 1161
remainderf, 1161
remainderl, 1161
remote_version, 1252
remove, 1047, 1434
forward_list, 810
list, 815
path, 1431
remove_all, 1431
remove_all_extents, 700
remove_const, 699
remove_copy, 1048
remove_copy_if, 1048
remove_cv, 699
remove_cvref, 701
remove_extent, 700
remove_filename
path, 1403
remove_if, 1047
forward_list, 810
remove_pointer, 700
remove_prefix
basic_string_view, 750
remove_reference, 699
remove_suffix
basic_string_view, 750
remove_volatile, 699
remquo, 1161
remquo, 1161

Index of library names 1726
remquol, 1161
rename, 1431, 1434
rend
  basic_string, 730
  basic_string_view, 749
  span, 876
rend(C&), 934
rend(initializer_list<E>), 934
rend(T (&array)[N]), 934
rep
  system_clock, 1205
replace, 1044
  basic_string, 736, 737
replace_copy, 1045
replace_copy_if, 1045
replace_extension
path, 1403
replace_filename
directory_entry, 1415
path, 1403
replace_if, 1044
required_alignment
  atomic_ref, 1484
  atomic_ref<floating-point>, 1484
  atomic_ref<integral>, 1484
  atomic_ref<T*>, 1484
reserve
  basic_string, 731, 1607
  unordered associative containers, 794
  vector, 819
reset
  any, 589
  bitset, 594
  optional, 572
  packaged_task, 1544
  shared_ptr, 631
  unique_ptr, 623, 625
  weak_ptr, 639
resetiosflags, 1358
resize
  basic_string, 730
  deque, 803
  forward_list, 809
  list, 813, 814
  valarray, 1152
  vector, 820
resize_file, 1432
resource
  polymorphic_allocator, 649
result
  local_info, 1255
result_of
  zombie, 466
result_of_t
  zombie, 466
resume
coroutine_handle<noop_coroutine_-_promise>,
  515
rethrow_exception, 501
rethrow_if_nested
  nested_exception, 502
rethrow_nested
  nested_exception, 502
return_temporary_buffer
  zombie, 466
reverse, 1050
  forward_list, 810
  list, 816
reverse_copy, 1051
reverse_iterator, 906
  base, 908
  basic_string, 723
  basic_string_view, 747
  constructor, 907, 908
  iter_move, 910
  iter_swap, 910
make_reverse_iterator non-member
  function, 910
  operator!-, 909
  operator*, 908
  operator+, 908, 910
  operator++, 908
  operator+=, 909
  operator->, 908
  operator-- , 908, 909
  operator<, 909
  operator<=, 909
  operator=, 908
  operator=, 909
  operator[], 908
reverse_view
  reverse_view, 985
rewind, 1434
rfind
  basic_string, 738
  basic_string_view, 752
riemann_zeta, 1175
riemann_zetaf, 1175
riemann_zetal, 1175
right, 1330
rint, 1161
rintf, 1161
rintl, 1161
root_directory
  path, 1405
root_name
  path, 1405
root_path
  path, 1405
rotate, 1051

Index of library names
rotate_copy, 1052
round, 1161
duration, 1199
time_point, 1204
round_error
numeric_limits, 483
round_indeterminate, 480
round_style
numeric_limits, 485
round_to_nearest, 480
round_toward_infinity, 480
round_toward_zero, 480
roundf, 1161
roundl, 1161
runtime_error, 530, 532
constructor, 532

s
lognormal_distribution, 1135
Same, 521
sample, 1052
save
sys_info, 1254
sbumpc
basic_streambuf, 1335
scalbln, 1161
scalbnf, 1161
scalbinl, 1161
scalbn, 1161
scalbnf, 1161
scalbln, 1161
scan_is
ctype, 1280
ctype<char>, 1283
scan_not
cctype, 1280
cctype<char>, 1283
scanf, 1434
SCHAR_MAX, 486
SCHAR_MIN, 486
scientific, 1330
scientific
chars_format, 711
SCNdFASTN, 1435
SCNdLEASTN, 1435
SCNdMAX, 1435
SCNdN, 1435
SCNdPTR, 1435
SCN fastest, 1435
SCN last, 1435
SCN MAX, 1435
SCN N, 1435
SCN PTR, 1435
SCN FASTN, 1435
SCN LAST, 1435
SCN MAX, 1435
SCN N, 1435
SCN PTR, 1435
scoped_allocator_adaptor, 655
allocate, 657
const_pointer, 655
const void_pointer, 655
construct, 657
constructor, 656, 657
deallocate, 657
destroy, 658
difference_type, 655
inner_allocator, 657
inner_allocator_type, 656
is always equal, 656
max_size, 657
operator!, 658
operator==, 658
outer_allocator, 657
outer_allocator_type, 655
pointer, 655
propagate on container copy assignment, 656
propagate on container move assignment, 656
propagate on container swap, 656
select on container swap, 656
select on container swap construction, 658
size_type, 655
value_type, 655
void_pointer, 655
scoped_lock, 1516
constructor, 1516
destructor, 1516
search, 1036–1038
search_n, 1037, 1038
second
local_info, 1255
second argument type
zombie, 466
seconds, 1177
time_of_day<sub-second duration>, 1248
time_of_day<seconds>, 1247
seed_seq, 1123
constructor, 1123, 1124
generate, 1124
param, 1124
size, 1124
SEEK_CUR, 1434
seek_dir
data sentiments

data sentiments

Index of library names
Index of library names

shared_future, 1539
  constructor, 1539
  destructor, 1539
  get, 1540
  operator=, 1539, 1540
  valid, 1540
  wait, 1540
  wait_for, 1540
  wait_until, 1540
shared_lock, 1520
  constructor, 1521, 1522
  destructor, 1522
  lock, 1522
  mutex, 1523
  operator bool, 1523
  operator=, 1522
  owns_lock, 1523
  release, 1523
  swap, 1523
  try_lock, 1522
  try_lock_for, 1523
  try_lock_until, 1522
  unlock, 1523
shared_mutex, 1513
shared_ptr, 627, 641, 1605
  atomic_compare_exchange_strong, 1606
  atomic_compare_exchange_strong🗹explicit, 1606
  atomic_compare_exchange_weak, 1606
  atomic_compare_exchange_weak🗹explicit, 1606
  atomic_exchange, 1606
  atomic_exchange🗹explicit, 1606
  atomic_load, 1606
  atomic_load🗹explicit, 1606
  atomic_store, 1606
  atomic_store🗹explicit, 1606
  const_pointer_cast, 637
  constructor, 629, 630
  destructor, 631
dynamic_pointer_cast, 637
  get, 631
  get_deleter, 637
  operator bool, 632
  operator!, 636
  operator*, 632
  operator->, 632
  operator<, 636
  operator<<, 638
  operator<=, 636
  operator=, 631
  operator==, 635, 636
  operator>, 636
  operator>>, 636
  operator[][], 632
  owner_before, 632
  reinterpret_pointer_cast, 637
reset, 631
  static_pointer_cast, 636
  swap, 631, 636
  use_count, 632
  shared_timed_mutex, 1514
shift
  valarray, 1151
  shift_left, 1053
  shift_right, 1054
showbase, 1329
showmanyc
  basic_filebuf, 1375
  basic_streambuf, 1337, 1375
showpoint, 1329
shovepos, 1329
shrink_to_fit
  basic_string, 731
  deque, 803
  vector, 819
SHRT_MAX, 486
SHRT_MIN, 486
shuffle, 1053
shuffle_order_engine, 1120
  constructor, 1121
  sig_atomic_t, 516
  SIG_DFL, 516
  SIG_ERR, 516
  SIG_IGN, 516
  SIGABRT, 516
  SIGFPE, 516
  SIGILL, 516
  SIGINT, 516
signal, 516
signaling_NaN
  numeric_limits, 484
signbit, 1161
SignedIntegral, 523
SIGSEGV, 516
SIGTERM, 516
sin, 1161
  complex, 1102
  valarray, 1154
sinf, 1161
single_view
  single_view, 952
sinh, 1161
  complex, 1102
  valarray, 1154
sinfh, 1161
sinhl, 1161
sinl, 1161
size
  array, 799, 800
  basic_string, 730
  basic_string_view, 750
bitset, 595
gslice, 1157
initializer_list, 503
match_results, 1463
### Index of library names

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed_seq</td>
<td>1124</td>
</tr>
<tr>
<td>single_view</td>
<td>952</td>
</tr>
<tr>
<td>slice</td>
<td>1155</td>
</tr>
<tr>
<td>span</td>
<td>876</td>
</tr>
<tr>
<td>subrange</td>
<td>949</td>
</tr>
<tr>
<td>valarray</td>
<td>1151</td>
</tr>
<tr>
<td>size(C&amp; c)</td>
<td>935</td>
</tr>
<tr>
<td>size(T (&amp;array)[N])</td>
<td>935</td>
</tr>
<tr>
<td>size_bytes</td>
<td></td>
</tr>
<tr>
<td>span</td>
<td>876</td>
</tr>
<tr>
<td>size_t</td>
<td>118, 473, 474, 756, 758, 1269, 1434</td>
</tr>
<tr>
<td>size_type</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>611</td>
</tr>
<tr>
<td>allocator_traits</td>
<td>610</td>
</tr>
<tr>
<td>basic_string</td>
<td>723</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>747</td>
</tr>
<tr>
<td>scoped_allocator_adaptor</td>
<td>655</td>
</tr>
<tr>
<td>SizedRange</td>
<td>943</td>
</tr>
<tr>
<td>SizedSentinel</td>
<td>893</td>
</tr>
<tr>
<td>skipws</td>
<td>1329</td>
</tr>
<tr>
<td>sleep_for</td>
<td></td>
</tr>
<tr>
<td>this_thread</td>
<td>1507</td>
</tr>
<tr>
<td>sleep_until</td>
<td></td>
</tr>
<tr>
<td>this_thread</td>
<td>1507</td>
</tr>
<tr>
<td>slice</td>
<td>1154</td>
</tr>
<tr>
<td>constructor</td>
<td>1154</td>
</tr>
<tr>
<td>size</td>
<td>1155</td>
</tr>
<tr>
<td>start</td>
<td>1155</td>
</tr>
<tr>
<td>stride</td>
<td>1155</td>
</tr>
<tr>
<td>slice_array</td>
<td>1155</td>
</tr>
<tr>
<td>operator==</td>
<td>1155</td>
</tr>
<tr>
<td>operator+=</td>
<td>1155</td>
</tr>
<tr>
<td>operator-=</td>
<td>1155</td>
</tr>
<tr>
<td>operator/=</td>
<td>1155</td>
</tr>
<tr>
<td>operator&lt;&lt;=</td>
<td>1155</td>
</tr>
<tr>
<td>operator&gt;=</td>
<td>1156</td>
</tr>
<tr>
<td>operator&gt;&gt;=</td>
<td>1155</td>
</tr>
<tr>
<td>operator%</td>
<td>1155</td>
</tr>
<tr>
<td>operator&amp;</td>
<td>1155</td>
</tr>
<tr>
<td>operator^</td>
<td>1155</td>
</tr>
<tr>
<td>operator</td>
<td></td>
</tr>
<tr>
<td>value_type</td>
<td>1155</td>
</tr>
<tr>
<td>snextc</td>
<td></td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1334</td>
</tr>
<tr>
<td>snprintf</td>
<td>1434</td>
</tr>
<tr>
<td>sort</td>
<td>1055</td>
</tr>
<tr>
<td>forward_list</td>
<td>810</td>
</tr>
<tr>
<td>list</td>
<td>816</td>
</tr>
<tr>
<td>sort_heap</td>
<td>1073</td>
</tr>
<tr>
<td>Sortable</td>
<td>903</td>
</tr>
<tr>
<td>space</td>
<td>1432</td>
</tr>
<tr>
<td>span</td>
<td>872</td>
</tr>
<tr>
<td>back</td>
<td>876</td>
</tr>
<tr>
<td>begin</td>
<td>876</td>
</tr>
<tr>
<td>cbegin</td>
<td>876</td>
</tr>
<tr>
<td>cend</td>
<td>876</td>
</tr>
<tr>
<td>constructor</td>
<td>874, 875</td>
</tr>
<tr>
<td>crbegin</td>
<td>876</td>
</tr>
<tr>
<td>crend</td>
<td>876</td>
</tr>
<tr>
<td>data</td>
<td>876</td>
</tr>
<tr>
<td>empty</td>
<td>876</td>
</tr>
<tr>
<td>end</td>
<td>876</td>
</tr>
<tr>
<td>first</td>
<td>875</td>
</tr>
<tr>
<td>front</td>
<td>876</td>
</tr>
<tr>
<td>last</td>
<td>875</td>
</tr>
<tr>
<td>rbegin</td>
<td>876</td>
</tr>
<tr>
<td>rend</td>
<td>876</td>
</tr>
<tr>
<td>size</td>
<td>876</td>
</tr>
<tr>
<td>size_bytes</td>
<td>876</td>
</tr>
<tr>
<td>subspace</td>
<td>949</td>
</tr>
<tr>
<td>subspace</td>
<td>943</td>
</tr>
<tr>
<td>SPDX</td>
<td></td>
</tr>
<tr>
<td>sph_bessel</td>
<td>1175</td>
</tr>
<tr>
<td>sph_besself</td>
<td>1175</td>
</tr>
<tr>
<td>sph_bessell</td>
<td>1175</td>
</tr>
<tr>
<td>sph_legendre</td>
<td>1175</td>
</tr>
<tr>
<td>sph_legendref</td>
<td>1175</td>
</tr>
<tr>
<td>sph_legendrel</td>
<td>1175</td>
</tr>
<tr>
<td>sph_neumann</td>
<td>1175</td>
</tr>
<tr>
<td>sph_neumannf</td>
<td>1175</td>
</tr>
<tr>
<td>sph_neumannl</td>
<td>1175</td>
</tr>
<tr>
<td>splice</td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>814, 815</td>
</tr>
<tr>
<td>splice_after</td>
<td></td>
</tr>
<tr>
<td>forward_list</td>
<td>809</td>
</tr>
<tr>
<td>split_view</td>
<td></td>
</tr>
<tr>
<td>split_view</td>
<td>978</td>
</tr>
<tr>
<td>sprintf</td>
<td>1434</td>
</tr>
<tr>
<td>sputbackc</td>
<td></td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1335</td>
</tr>
<tr>
<td>sputc</td>
<td></td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1335</td>
</tr>
<tr>
<td>sputn</td>
<td></td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1335</td>
</tr>
<tr>
<td>sqrt</td>
<td>1161</td>
</tr>
<tr>
<td>complex</td>
<td>1102</td>
</tr>
<tr>
<td>valarray</td>
<td>1154</td>
</tr>
<tr>
<td>sqrtf</td>
<td>1161</td>
</tr>
<tr>
<td>sqrt1</td>
<td>1161</td>
</tr>
<tr>
<td>srand</td>
<td>474, 1142</td>
</tr>
<tr>
<td>sscanf</td>
<td>1434</td>
</tr>
<tr>
<td>ssize(C&amp; c)</td>
<td>935</td>
</tr>
<tr>
<td>ssize(T (&amp;array)[N])</td>
<td>935</td>
</tr>
<tr>
<td>stable_partition</td>
<td>1063</td>
</tr>
<tr>
<td>stable_sort</td>
<td>1055</td>
</tr>
<tr>
<td>stack</td>
<td>870</td>
</tr>
<tr>
<td>constructor</td>
<td>871</td>
</tr>
<tr>
<td>swap</td>
<td>871</td>
</tr>
<tr>
<td>start</td>
<td></td>
</tr>
<tr>
<td>galice</td>
<td>1157</td>
</tr>
<tr>
<td>slice</td>
<td>1155</td>
</tr>
<tr>
<td>starts_with</td>
<td></td>
</tr>
<tr>
<td>basic_string</td>
<td>740</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>751, 752</td>
</tr>
<tr>
<td>state</td>
<td></td>
</tr>
<tr>
<td>fpos</td>
<td>1324</td>
</tr>
<tr>
<td>wbuffer_convert</td>
<td>1611</td>
</tr>
<tr>
<td>wstring_convert</td>
<td>1610</td>
</tr>
<tr>
<td>state_type</td>
<td></td>
</tr>
<tr>
<td>char_traits</td>
<td>716</td>
</tr>
</tbody>
</table>
Index of library names

wbuffer_convert, 1612
wstring_convert, 1610
static_pointer_cast
shared_ptr, 636
status, 1432
directory_entry, 1417
status_known, 1433
stddev
normal_distribution, 1134
stderr, 1434
stdin, 1434
stdout, 1434
steady_clock, 1212
stem
path, 1405
stod, 744, 745
stof, 744, 745
stoi, 744
stol, 744
stoll, 744
store
atomic, 1490
atomic<floating-point>, 1490
atomic<integral>, 1490
atomic<shared_ptr<T>>, 643
atomic<T>, 1490
atomic<weak_ptr<T>>, 644
atomic_ref, 1484
atomic_ref<floating-point>, 1484
atomic_ref<integral>, 1484
atomic_ref<T>, 1484
stossc
zombie, 466
stoul, 744
stoull, 744
str
basic_istringstream, 1368
basic_osstringstream, 1370
basic_stringbuf, 1365
basic_stringstream, 1371
istreamstream, 1602
match_results, 1463
ostreamstream, 1603
strstream, 1604
strstreambuf, 1599
sub_match, 1456
strcat, 756
strchr, 756
strcspn, 756
strcspn, 756
streambuf, 1312, 1331
streamoff, 1316, 1324
streampos, 1312
streamsize, 1317
strerror, 756
strftime, 1269, 1302
StrictTotallyOrdered, 527
stride
gslice, 1157
slice, 1155
string
operator"s", 745
path, 1404
string_view
operator"sv", 755
stringbuf, 1312, 1362
stringstream, 1312, 1362
strlen, 756, 1598, 1602
strncat, 756
strncpy, 756
strong_equal, 511
strong_equality, 505
equal, 505
equivalent, 505
nonequal, 505
nonequivalent, 505
operator weak_equality, 505
operator!=, 506
operator<>, 506
operator==, 505
strong_order, 510
strong_ordering, 508
equal, 508
equivalent, 508
greater, 508
less, 508
operator partial_ordering, 509
operator strong_equality, 509
operator weak_equality, 509
operator weak_ordering, 509
operator!=, 509
operator<, 509
operator<>, 509
operator==, 509
operator>, 509
operator>=, 509
strpbrk, 756
strrchr, 756
strspn, 756
strstr, 756
strstream, 1603
constructor, 1603
destructor, 1603, 1604
freeze, 1604
pcount, 1604
rdbuf, 1604
str, 1604
strstreambuf, 1596
constructor, 1598, 1599
destructor, 1599
freeze, 1599
overflow, 1599
pbackfail, 1600
Index of library names

any, 589, 590
array, 800
basic_filebuf, 1374
basic_fstream, 1383
basic_ifstream, 1379
basic_ios, 1328
basic_iostream, 1351
basic_istream, 1342
basic_iostream, 1368
basic_ofstream, 1381
basic_ostringstream, 1370
basic_ostringstream, 1371
basic_string, 737, 742
basic_string_view, 750
basic_stringbuf, 1365
basic_stringstream, 1365
basic_stringstream, 1371
basic_syncbuf, 1386
function, 677, 678
match_results, 1464, 1465
optional, 570, 574
packaged_task, 1543, 1544
pair, 551
path, 1403, 1408
priority_queue, 1535, 1536
queue, 867
shared_lock, 1523
shared_ptr, 631, 636
stack, 871
thread, 1505, 1506
tuple, 558
unique_lock, 1520
unique_ptr, 623
valarray, 1151, 1154
variant, 582, 585
vector, 819
vector<bool>, 823
weak_ptr, 639, 640
swap(unique_ptr&, unique_ptr&), 626
swap_ranges, 1042
Swappably, 524
SwappablyWith, 524
swprintf, 756
swscanf, 756
symlink_status, 1433
directory_entry, 1417
directory, 1417
sync
basic_filebuf, 1377
basic_istream, 1349
basic_streambuf, 1337
basic_syncbuf, 1386
sync_with_stdio
ios_base, 1322
syncbuf, 1312, 1384
synchronized_pool_resource, 650
constructor, 652
destructor, 652

pcount, 1599
seekoff, 1600
seekpos, 1601
setbuf, 1601
str, 1599
underflow, 1600
strtof, 474
strtol, 474
strtold, 474
strtol, 474
strtoimax, 1435
strtok, 756
strtol, 474
strtof, 474
strtoim, 1435
strxfrm, 756
student_t_distribution, 1137
constructor, 1138
mean, 1138
sub_match, 1456
compare, 1456
constructor, 1456
length, 1456
operator basic_string, 1456
operator!=, 1457–1460
operator<, 1457–1460
operator<, 1457–1460
operator<=, 1457–1460
operator==, 1457–1460
operator>, 1457–1460
operator>==, 1457–1460
operator>=, 1457–1460
str, 1456
subrange, 947
subrange, 949
subseconds

time_of_day<sub-second duration>, 1248
subspan
span, 875
substr
basic_string, 739
basic_string_view, 751
subtract_with_carry_engine, 1117
constructor, 1118
suffix
match_results, 1463
sum
valarray, 1151
sungec
basic_streambuf, 1335
suspend_always, 515
await_ready, 515
await_resume, 515
suspend_never, 515
await_ready, 515
await_resume, 515
swap, 546, 562
do_allocate, 652
do_deallocate, 652
do_is_equal, 653
options, 652
release, 652
upstream_resource, 652

syntax_option_type, 1446
awk, 1446, 1447
basic, 1446, 1447
collate, 1446, 1447, 1476
ECMAScript, 1446, 1447
egrep, 1446, 1447
extended, 1446, 1447
icase, 1446, 1447
multiline, 1447	nosubs, 1446, 1447
optimize, 1446, 1447

sys_days, 1177
operator<<, 1205

sys_info, 1254
abbrev, 1254
begin, 1254
dd, 1254
offset, 1254
operator<<, 1254
save, 1254
sys_seconds, 1177
sys_time, 1177

from_stream, 1209
operator<<, 1209
take_view
take_view, 970
tan, 1161
tanf, 1161
tanh, 1161
tanhf, 1161
tanh, 1161
tanl, 1161
tanl, 1161
tan<
function, 677
link, 1263
target_type
function, 677
tellg
tell

from basic_istream, 1349
tellp
tellp

from basic_ostream, 1353
temp_directory_path, 1433
terminate, 489, 500
terminate_handler, 468, 500
test

bitset, 595
test_and_set

atomic_flag, 1498
tgamma, 1161
tgammaf, 1161
tgammal, 1161
this_thread

gid, 1506
sleep_for, 1507
sleep_until, 1507
yield, 1506

thousands_sep

moneypunct, 1307
numpunct, 1295

thread, 1503

destructor, 1505
detach, 1506
gid, 1506
hardware_concurrency, 1506

join, 1505
joinable, 1505
operator=, 1505
swap, 1505, 1506

thread::id, 1503

operator!=, 1504
operator<, 1504
operator<=, 1504
operator=, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<, 1504
operator<, 1504
operator=, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator<<, 1504
operator==, 1504
operator>, 1504
operator>=, 1504
throw_with_nested
nested_exception, 502
tie, 559
basic_ios, 1326
time, 1269
time_get, 1298
date_order, 1299
do_date_order, 1300
do_get, 1301
do_get_date, 1300
do_get_monthname, 1300
do_get_time, 1300
do_get_weekday, 1300
do_get_year, 1301
get, 1299
get_date, 1299
get_monthname, 1299
get_time, 1299
get_weekday, 1299
get_year, 1299
time_get_byname, 1301
time_of_day, 1245
time_of_day<hours>, 1245
compiler, 1248
hours, 1248
make12, 1248
make24, 1248
minutes, 1248
operator precision, 1248
operator<<, 1250
seconds, 1248
subseconds, 1248
to_duration, 1248
time_of_day<minutes>, 1246
compiler, 1246
hours, 1246
make12, 1246
make24, 1246
operator precision, 1246
operator<<, 1248
to_duration, 1246
time_of_day<seconds>, 1247
compiler, 1247
hours, 1247
make12, 1247
make24, 1247
operator precision, 1247
operator<<, 1249
seconds, 1247
to_duration, 1247
time_point, 1201
ceil, 1204
compiler, 1202
floor, 1204
max, 1203
min, 1203
operator!=, 1203
operator+, 1203
operator++, 1202
operator+=, 1203
operator-, 1203
operator--, 1203
operator-=, 1203
operator<, 1204
operator<=, 1204
operator==, 1203
operator>, 1204
operator>=, 1204
operator>-, 1204
round, 1204
time_point_cast, 1204
time_since_epoch, 1202

Index of library names 1735
time_of_day<sub-second duration>, 1248
time_of_day<hours>, 1246
time_of_day<minutes>, 1246
time_of_day<seconds>, 1247
to_integer
    byte, 477
to_local
    time_zone, 1256
to_stream
    day, 1218
duration, 1201
file_time, 1212
gps_time, 1211
local_time, 1213
month, 1220
month_day, 1228
sys_time, 1206
tai_time, 1209
utc_time, 1207
weekday, 1225
year, 1223
year_month, 1232
year_month_day, 1235
zoned_time, 1261
to_string, 744
    bitset, 595
to_sys
    time_zone, 1256
utc_clock, 1207
to_time_t
    system_clock, 1205
to_ullong
    bitset, 595
to_ulong
    bitset, 595
to_utc
    gps_clock, 1210
tai_clock, 1208
to_wstring, 745
tolower, 755, 1279
cctype, 1280
cctype<char>, 1283
toupper, 755, 1278
cctype, 1280
cctype<char>, 1283
towctrans, 755
towlower, 755
towupper, 755
traits_type
    basic_string, 723
    basic_string_view, 747
transform, 1042
collate, 1297
regex_traits, 1450
transform_exclusive_scan, 1089
transform_inclusive_scan, 1090
transform_primary
    regex_traits, 1450
transform_reduce, 1086, 1087
transform_view
    transform_view, 964
translate
    regex_traits, 1450
translate_nocase
    regex_traits, 1450
traps
    numeric_limits, 484
treat_asFloatingPoint, 1193
treat_asFloatingPoint_v, 1177
true_type, 688
truename
    numpunct, 1296
trunc, 1161
truncf, 1161
truncl, 1161
try_emplace
    map, 830
    unordered_map, 848, 849
try_lock, 1523
    shared_lock, 1522
    unique_lock, 1519
try_lock_for
    shared_lock, 1523
    unique_lock, 1519
try_lock_until
    shared_lock, 1522
    unique_lock, 1519
try_to_lock, 1515
try_to_lock_t, 1515
tuple, 553, 554, 801
    constructor, 556, 557
    forward_as_tuple, 559
    get, 561
    make_tuple, 558
    operator!, 562
    operator<, 562
    operator<=, 557, 558
    operator==, 561
    operator>, 562
    operator>=, 562
    swap, 558
    tie, 559
tuple_cat, 559
tuple_element, 552, 560, 801, 877
tuple_size, 552, 560, 801, 877
    in general, 560
type
    any, 590
    file_status, 1413
type_identity, 701
type_identity_t, 682
type_index, 708
    constructor, 708
    hash_code, 709
    name, 709
    operator!, 709
    operator<, 709
Index of library names

operator<=, 709
operator==, 708
operator>, 709
operator>=, 709
type_info, 111, 496, 497
    before, 497
    hash_code, 497
    name, 497
    operator!=, 497
    operator==, 497
tzdb, 1250
    current_zone, 1250
    locate_zone, 1250
tzdb_list, 1251
    begin, 1251
    cbegin, 1251
    cend, 1251
    end, 1251
    erase_after, 1251
    front, 1251
u16streampos, 1312
u16string
    operator"s, 745
    path, 1404
u16string_view
    operator"sv, 755
u32streampos, 1312
u32string
    operator"s, 746
    path, 1404
u32string_view
    operator"sv, 755
u8path, 1612
u8string
    operator"s, 745
    path, 1404
u8string_view
    operator"sv, 755
UCHAR_MAX, 486
uflow
    basic_filebuf, 1376
    basic_streambuf, 1338
uint16_t, 487
uint32_t, 487
uint64_t, 487
uint8_t, 487
uint_fast16_t, 487
uint_fast32_t, 487
uint_fast64_t, 487
uint_fast8_t, 487
uint_least16_t, 487
uint_least32_t, 487
uint_least64_t, 487
UINT_MAX, 486
uintmax_t, 487
uintptr_t, 487
ULLONG_MAX, 486
ULONGLONG_MAX, 486
unary_function
    zombie, 466
unary_negate
    zombie, 466
uncaught_exception
    zombie, 466
uncaught_exceptions, 426, 501
undeclare_no_pointers, 606
undeclare_reachable, 605
underflow
    basic_filebuf, 1375
    basic_streambuf, 1337
    basic_stringbuf, 1365
    stringstreambuf, 1600
underflow_error, 530, 533
    constructor, 533
underlying_type, 701
unexpected
    zombie, 466
unexpected_handler
    zombie, 466
unget
    basic_istream, 1349
ungetc, 1434
ungetwc, 756
uniform_int_distribution, 1125
    a, 1126
    b, 1126
    constructor, 1126
uniform_real_distribution, 1126
    a, 1127
    b, 1127
    constructor, 1127
uninitialized_construct_using_allocator, 609
uninitialized_copy, 614, 615
uninitialized_copy_n, 615
uninitialized_default_construct, 613
uninitialized_default_construct_n, 613, 614
uninitialized_fill, 616
uninitialized_fill_n, 617
uninitialized_move, 615
uninitialized_move_n, 616
uninitialized_value_construct, 614
uninitialized_value_construct_n, 614
unique, 1049
    forward_list, 810
    list, 815
    local_info, 1255
unique_copy, 1049
unique_lock, 1516
    constructor, 1517, 1518
destructor, 1518
lock, 1518
mutex, 1520
operator bool
    basic_stringbuf, 1365
    stringstreambuf, 1600
operator=
    basic_filebuf, 1376
    basic_streambuf, 1338
operator bool
    basic_stringbuf, 1365
operator=
    basic_filebuf, 1376
    basic_streambuf, 1338
Index of library names

load_factor, 794
local_iterator, 785
mapped_type, 784
max_bucket_count, 793
max_load_factor, 794
merge, 791
node_type, 785
operator[], 848
rehash, 794
reserve, 794
try_emplace, 848, 849
value_type, 784
unordered_multimap, 841, 850
begin, 794
bucket, 793
bucket_count, 793
bucket_size, 793
cbegin, 794
cend, 794
clear, 791
const_local_iterator, 785
constructor, 785, 854
contains, 792
count, 792
eplace, 787
eplace_hint, 788
end, 794
equal_range, 793
erase, 791
extract, 790
find, 791
hash_function, 787
hasher, 784
insert, 788, 854
key_eq, 787
key_equal, 784
key_type, 784
load_factor, 794
local_iterator, 785
mapped_type, 784
max_bucket_count, 793
max_load_factor, 794
merge, 791
node_type, 785
operator[], 848
rehash, 794
reserve, 794
try_emplace, 848, 849
value_type, 784
unordered_multiset, 842, 859
begin, 794
bucket, 793
bucket_count, 793
bucket_size, 793
cbegin, 794
cend, 794
clear, 791
const_local_iterator, 785
constructor, 785, 863
contains, 792
count, 792
emplace, 787
emplace_hint, 788
end, 794
equal_range, 793
erase, 791
extract, 790
find, 791
hash_function, 787
hasher, 784
insert, 788
key_eq, 787
key_equal, 784
key_type, 784
load_factor, 794
local_iterator, 785
mapped_type, 784
max_bucket_count, 793
max_load_factor, 794
merge, 791
node_type, 785
rehash, 794
reserve, 794
value_type, 784
unordered_set, 842, 855
begin, 794
bucket, 793
bucket_count, 793
bucket_size, 793
cbegin, 794
cend, 794
clear, 791
const_local_iterator, 785
consstructor, 785, 858, 859
contains, 792
count, 792
emplace, 787
emplace_hint, 788
end, 794
equal_range, 793
erase, 791
extract, 790
find, 791
hash_function, 787
hasher, 784
insert, 788
key_eq, 787
key_equal, 784
key_type, 784
load_factor, 794
local_iterator, 785
mapped_type, 784
max_bucket_count, 793
max_load_factor, 794
merge, 791
node_type, 785
rehash, 794
reserve, 794
value_type, 784
unreachable_sentinel_t, 927
operator!=, 928
operator==, 928
unsetf
ios_base, 1321
unshift
codecvt, 1285
UnsignedIntegral, 523
unsynchronized_pool_resource, 650
constructor, 652
destructor, 652
do_allocate, 652
do_deallocate, 652
do_is_equal, 653
options, 652
release, 652
upstream_resource, 652
unwrap_ref_decay, 658
unwrap_ref_decay_t, 658
unwrap_reference, 663
upper_bound, 1060
ordered associative containers, 781
uppercase, 1329
upstream_resource
monotonic_buffer_resource, 654
synchronized_pool_resource, 652
unsynchronized_pool_resource, 652
use_count
shared_ptr, 632
weak_ptr, 640
use_facet
locale, 1278
uses_allocator, 607
promise, 1534
uses_allocator<tuple>, 562
uses_allocator_construction_args, 607, 608
USHRT_MAX, 486
utc_clock, 1206
from_sys, 1207
now, 1206
to_sys, 1207
utc_seconds, 1177
utc_time, 1177
from_stream, 1208
operator<<, 1207
to_stream, 1207
va_arg, 515
va_copy, 515
va_end, 467, 515
va_list, 467, 515
va_start, 515, 516
valarray, 1145, 1158
apply, 1151
constructor, 1147
cshift, 1151
destructor, 1147
max, 1151
min, 1151
Index of library names

valid
  future, 1538
  packaged_task, 1543
  shared_future, 1540
value
  error_code, 540
  error_condition, 541
  optional, 571
  regex_traits, 1451
value_comp
  ordered associative containers, 775
value_compare
  ordered associative containers, 774
value_or
  optional, 571, 572
value_type
  allocator, 611
  atomic, 1488
  atomic_ref, 1483
  basic_string, 723
  basic_string_view, 747
  complex, 1097
  gslice_array, 1157
  indirect_array, 1159
  integer_sequence, 1159
  integral_constant, 688
  mask_array, 1158
  optional, 564
  ordered associative containers, 774
  path, 1397
  polymorphic_allocator, 647
  scoped_allocator_adaptor, 655
  slice_array, 1155
  split_view::outer_iterator::value_type, 980
  unordered associative containers, 784

valueless_by_exception
  variant, 582

variant, 577
  constructor, 578, 579
  destructor, 579
  emplace, 581
  get, 583
  get_if, 583, 584
  holds_alternative, 583
  index, 582
  operator!=, 584
  operator<, 584
  operator<=, 584
  operator==, 580
  operator>=, 584
  operator>, 584
  operator>=, 584
  swap, 582
  valueless_by_exception, 582
  visit, 585

variant_alternative, 583
variant_size, 582, 583

vector, 816
  constructor, 818, 819
  operator<, 818
  operator==, 818

vector<bool>, 821
  flip, 822
  swap, 823
  vfprintf, 1434
  vscanf, 1434
  vfwprintf, 756
  vfwscanf, 756

View, 944
  view_interface, 945
  back, 947
  front, 947
  ViewableRange, 945
  visit, 585

void_pointer
  allocator_traits, 610
  scoped_allocator_adaptor, 655
  vfprintf, 1434
  vscanf, 1434

Index of library names 1740
vsnprintf, 1434
vsprintf, 1434
vsscanf, 1434
vswprintf, 756
vswscanf, 756
vwprintf, 756
vwscanf, 756

wait
  condition_variable, 1527
  condition_variable_any, 1530
  future, 1538
  shared_future, 1540
wait_for
  condition_variable, 1528, 1529
  condition_variable_any, 1531
  future, 1538
  shared_future, 1540
wait_until
  condition_variable, 1527, 1528
  condition_variable_any, 1530, 1531
  future, 1538
  shared_future, 1540
wbuffer_convert, 1611
  constructor, 1612
destructor, 1612
rdbuf, 1612
state, 1611
state_type, 1612
wcerr, 1315
WCHAR_MAX, 756
WCHAR_MIN, 756
wcin, 1315
wclog, 1315
wcout, 1315
wcrstombs, 758
wctomb, 474, 758
wcrtomb, 756, 758
wcscat, 756
wcschr, 756
wcscmp, 756
wcscoll, 756
wcscpy, 756
wcscspn, 756
wcsftime, 756
wcslen, 756
wcsncat, 756
wcsncmp, 756
wcsncpy, 756
wcspbrk, 756
wcsrchr, 756
wcsrtombs, 756
wcsxfrm, 756
wcsxfrm, 756
wcstod, 756
wcstof, 756
wcstoi, 756
wcstol, 756
wcstold, 756

wcstoll, 756
wcstombs, 474, 758
wcstoul, 756
wcstoull, 756
wcstoumax, 1435
wcsxfrm, 756
wctob, 756
wctomb, 474, 758
wctrans, 755
wctrans_t, 755
wchar_t, 755
weak_equal, 504
weak_equal, 504
equivalent, 504
nonequivalent, 504
operator!=, 505
operator<, 505
operator==, 505

weak_from_this
  enable_shared_from_this, 641
weak_order, 510
weak_ordering, 507
equivalent, 507
greater, 507
less, 507
operator partial_ordering, 508
operator weak_equality, 508
operator!=, 508
operator<, 508
operator<=, 508
operator>=, 508
operator>, 508
operator>==, 508
weak_ptr, 630, 638, 641
  constructor, 639
destructor, 639
expired, 640
lock, 640
operator=, 639
owner_before, 640
reset, 639
swap, 639, 640
use_count, 640

weakly_canonical, 1434
WeaklyIncrementable, 892
weekday, 1223
  constructor, 1224
  from_stream, 1225
  ok, 1224
operator unsigned, 1224
operator+, 1225
operator++, 1224
operator+=, 1224
operator-, 1225
operator--, 1224
operator-=, 1224
operator<<, 1225
Index of library names
Index of library names
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, 439
additional execution policies supported by parallel algorithms, 710, 990
additional file_type enumerators for file systems supporting additional types of file, 1410
additional formats for time_get::do_get_date, 1300
additional supported forms of preprocessing directive, 428
algorithms for producing the standard random number distributions, 1125
alignment, 59
alignment additional values, 60
allocation of bit-fields within a class object, 265
any use of an invalid pointer other than to perform indirection or deallocate, 56
argument values to construct basic_ios::failure, 1328
assignability of placeholder objects, 674
behavior of iostream classes when traits::pos_type is not streampos or when traits::off_type is not streamoff, 1312
behavior of non-standard attributes, 223
behavior of strstreambuf::setbuf, 1601
bits in a byte, 50
choice of larger or smaller value of floating literal, 19
concatenation of some types of string literals, 21
consistency of build level and violation continuation mode, 29
conversions between pointers and integers, 113
converting characters from source character set to execution character set, 9
converting function pointer to object pointer and vice versa, 114
default configuration of a pool, 652
default next_buffer_size for a monotonic_buffer_resource, 654
default number of buckets in unordered_map, 848
default number of buckets in unordered_multimap, 854
default number of buckets in unordered_multiset, 863
default number of buckets in unordered_set, 859
defining main in freestanding environment, 77
definition and meaning of __STDC__, 441, 1576
definition and meaning of __STDC_VERSION__, 441
definition of NULL, 475, 1592
derived type for typeid, 111
diagnostic message, 3
dynamic initialization of static inline variables before main, 79
dynamic initialization of static variables before main, 78, 79
dynamic initialization of thread-local variables before entry, 79
effect of calling associated Laguerre polynomials with \( n \geq 128 \) or \( m \geq 128 \), 1171
effect of calling associated Legendre polynomials with \( l \geq 128 \), 1171
effect of calling basic_filebuf::setbuf with nonzero arguments, 1376
effect of calling basic_filebuf::sync when a get area exists, 1377
effect of calling basic_streambuf::setbuf with nonzero arguments, 1367
effect of calling cylindrical Bessel functions of the first kind with \( \nu \geq 128 \), 1172
effect of calling cylindrical Neumann functions with \( \nu \geq 128 \), 1173
effect of calling Hermite polynomials with \( n \geq 128 \), 1174
effect of calling ios_base::sync_with_stdio after any input or output operation on standard streams, 1322
effect of calling irregular modified cylindrical Bessel functions with \( \nu \geq 128 \), 1173
effect of calling Laguerre polynomials with \( n \geq 128 \), 1174
effect of calling Legendre polynomials with \( l \geq 128 \), 1174
effect of calling regular modified cylindrical Bessel functions with \( \nu \geq 128 \), 1172
effect of calling spherical associated Legendre functions with \( l \geq 128 \), 1175
effect of calling spherical Bessel functions with \( n \geq 128 \), 1176
effect of calling spherical Neumann functions with \( n \geq 128 \), 1176
effect of filesystem::copy, 1423
effect on C locale of calling locale::global, 1278
encoding of universal character name not in execution character set, 18
error_category for errors originating outside the operating system, 472
establishing of and argument for violation handler, 228
exception type when random_device constructor fails, 1123
exception type when random_device::operator() fails, 1123
exception type when shared_ptr constructor fails, 629, 630
exceptions thrown by standard library functions that have a potentially-throwing exception specification, 472
execution character set and execution wide-character set, 11
exit status, 489
extended signed integer types, 65
file type of the file argument of filesystem::status, 1433
formatted character sequence generated by time_put::do_put in C locale, 1302
forward progress guarantees for implicit threads of parallel algorithms (if not defined for thread), 989
growth factor for monotonic_buffer_resource, 654
headers for freestanding implementation, 455
how random_device::operator() generates values, 1123
how the set of importable headers is determined, 236
interactive device, 7
interpretation of the path character sequence with format path::auto_format, 1410
largest supported value to configure the largest allocation satisfied directly by a pool, 652
largest supported value to configure the maximum number of blocks to replenish a pool, 652
linkage of main, 77
linkage of names from C standard library, 456
linkage of objects between C++ and other languages, 222
locale names, 1276
lvalue-to-rvalue conversion of an invalid pointer value, 86
manner of search for included source file, 431
mapping from name to catalog when calling messages::do_open, 1308
mapping from physical source file characters to basic source character set, 10, 1583
mapping header name to header or external source file, 13
mapping of pointer to integer, 113
mapping physical source file characters to basic source character set, 9
mapping to message when calling messages::do_get, 1309
maximum depth of recursive template instantiations, 391
maximum size of an allocated object, 120, 495
meaning of ’, \, /, *, or // in a q-char-sequence or an h-char-sequence, 13
meaning of asm declaration, 219
meaning of attribute declaration, 152
meaning of dot-dot in root-directory, 1398
mechanism for selecting the build level, 228
negative value of character literal in preprocessor, 430
nesting limit for #include directives, 431
NTCTS in basic_ostream<charT, traits>& operator<<(nullptr_t), 1356
number of placeholders for bind expressions, 659, 674
number of threads in a program under a freestanding implementation, 72
numeric values of character literals in #if directives, 430
operating system on which implementation depends, 1390
parameters to main, 77
passing argument of class type through ellipsis, 108
physical source file characters, 9
presence and meaning of native_handle_type and native_handle, 1500
range defined for character literals, 18
rank of extended signed integer type, 69
required alignment for atomic_ref type’s operations, 1483, 1485–1487
required libraries for freestanding implementation, 6
resource limits on a message catalog, 1309
result of filesystem::file_size, 1428
result of inexact floating-point conversion, 88
return value of bad_alloc::what, 495
return value of bad_any_cast::what, 586
return value of bad_array_new_length::what, 495
return value of bad_cast::what, 497
return value of bad_exception::what, 500
return value of bad_function_call::what, 675
return value of bad_optional_access::what, 572
return value of bad_typeid::what, 498
return value of bad_variant_access::what, 586
return value of bad_weak_ptr::what, 627
return value of char_traits<char16_t>::eof, 718
Index of implementation-defined behavior
return value of `char_traits<char32_t>::eof`, 718
return value of `exception::what`, 499
return value of `type_info::name()`, 497
search locations for " header, 431
search locations for <> header, 431
semantics of an access through a volatile glvalue, 161
semantics of linkage specification on templates, 336
semantics of linkage specifiers, 220
semantics of non-standard escape sequences, 18
semantics of parallel algorithms invoked with implementation-defined execution policies, 990
semantics of `token` parameter and default token value used by `random_device` constructors, 1123
sequence of places searched for a header, 431
set of character types that iostreams templates can be instantiated for, 1275, 1312
signedness of `char`, 162
`sizeof` applied to fundamental types other than `char`, `signed char`, and `unsigned char`, 118
source location of precondition violation, 228
stack unwinding before call to `std::terminate`, 422, 426
start-up and termination in freestanding environment, 77
string resulting from `_func_`, 199
support for extended alignments, 60
supported multibyte character encoding rules, 717
supported `root-names` in addition to any operating system dependent `root-names`, 1397, 1398
text of `_DATE_` when date of translation is not available, 439
text of `_TIME_` when time of translation is not available, 440
threads and program points at which deferred dynamic initialization is performed, 78, 79
type of a directory-like file, 1417, 1419
type of `array::const_iterator`, 799
type of `array::iterator`, 799
type of `basic_string::const_iterator`, 723
type of `basic_string::iterator`, 723
type of `basic_string_view::const_iterator`, 747, 749
type of `default_random_engine`, 1122
type of `deque::const_iterator`, 801
type of `deque::iterator`, 801
type of `forward_list::const_iterator`, 805
type of `forward_list::iterator`, 805
type of `list::const_iterator`, 811
type of `list::iterator`, 811
type of `map::const_iterator`, 826
type of `map::iterator`, 826
type of `multimap::const_iterator`, 831
type of `multimap::iterator`, 831
type of `multiset::const_iterator`, 838
type of `multiset::iterator`, 838
type of `ptrdiff_t`, 127, 476
type of `regex_constants::error_type`, 1448
type of `regex_constants::match_flag_type`, 1447
type of `set::const_iterator`, 835
type of `set::iterator`, 835
type of `size_t`, 476
type of `span::const_iterator`, 872
type of `span::iterator`, 872
type of `syntax_option_type`, 1446
type of `unordered_map::const_iterator`, 844
type of `unordered_map::const_local_iterator`, 844
type of `unordered_map::iterator`, 844
type of `unordered_multimap::const_iterator`, 850
type of `unordered_multimap::const_local_iterator`, 850
type of `unordered_multimap::iterator`, 850
type of `unordered_multiset::const_iterator`, 860
type of `unordered_multiset::const_local_iterator`, 860
type of `unordered_multiset::iterator`, 860
type of `unordered_multiset::local_iterator`, 860
type of `unordered_set::const_iterator`, 855
type of `unordered_set::const_local_iterator`, 855
type of `unordered_set::iterator`, 855
type of `unordered_set::local_iterator`, 855
type of `vector::const_iterator`, 817
type of `vector::iterator`, 817
type of `vector<bool>::const_iterator`, 821
type of `vector<bool>::iterator`, 821
underlying type for enumeration, 206
underlying type of `bool`, 66
underlying type of `char`, 66
underlying type of `wchar_t`, 66
value of bit-field that cannot represent assigned value, 134
incremented value, 109
initializer, 186

Index of implementation-defined behavior
value of character literal outside range of corresponding type, 18
value of `ctype<char>::table_size`, 1283
value of `has-attribute-expression` for non-standard attributes, 429
value of multicharacter literal, 17
value of `pow(0,0)`, 1102
value of result of inexact integer to floating-point conversion, 88
value of wide-character literal containing multiple characters, 18
value of wide-character literal with single c-char that is not in execution wide-character set, 18
value representation of floating-point types, 66
value representation of pointer types, 67
values of a trivially copyable type, 64
values of various `ATOMIC_..._LOCK_FREE` macros, 1482
whether `<cfenv>` functions can be used to manage floating-point status, 1095
whether a given `atomic` type’s operations are always lock free, 642, 644, 1489, 1490, 1492, 1494, 1495
whether a given `atomic_ref` type’s operations are always lock free, 1483, 1485–1487
whether an implementation has relaxed or strict pointer safety, 59
whether functions from Annex K of the C standard library are declared when C++ headers are included, 454
whether `get_pointer_safety` returns `pointer_safety::relaxed` or `pointer_safety::preferred` if the implementation has relaxed pointer safety, 606
whether locale object is global or per-thread, 1274
whether pragma `FENV_ACCESS` is supported, 1095
whether `rand` may introduce a data race, 1142
whether sequence pointers are copied by `basic_filebuf` move constructor, 1373
whether sequence pointers are copied by `basic_stringbuf` move constructor, 1364
whether sequence pointers are initialized to null pointers, 1364
whether source of translation units must be available to locate template definitions, 10
whether stack is unwound before calling the function `std::terminate` when a `noexcept` specification is violated, 426
whether the implementation is hosted or freestanding, 455
whether the lifetime of a parameter ends when the callee returns or at the end of the enclosing full-expression, 107
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, 10
whether the thread that executes `main` and the threads created by `std::thread` provide concurrent forward progress guarantees, 76
whether `time_get::do_get_year` accepts two-digit year numbers, 1301
whether values are rounded or truncated to the required precision when converting between `time_t` values and `time_point` objects, 1205
which functions in the C++ standard library may be recursively reentered, 470
which non-standard-layout objects containing no data are considered empty, 52
which scalar types have unique object representations, 696
width of integral type, 65