Working Draft, Standard for Programming Language C++

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

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.3</td>
<td>Header <code>&lt;iterator&gt;</code> synopsis</td>
<td>860</td>
</tr>
<tr>
<td>27.4</td>
<td>Iterator primitives</td>
<td>863</td>
</tr>
<tr>
<td>27.5</td>
<td>Iterator adaptors</td>
<td>865</td>
</tr>
<tr>
<td>27.6</td>
<td>Stream iterators</td>
<td>876</td>
</tr>
<tr>
<td>27.7</td>
<td>Range access</td>
<td>882</td>
</tr>
<tr>
<td>27.8</td>
<td>Container access</td>
<td>883</td>
</tr>
<tr>
<td>28.1</td>
<td>General</td>
<td>884</td>
</tr>
<tr>
<td>28.2</td>
<td>Header <code>&lt;algorithm&gt;</code> synopsis</td>
<td>884</td>
</tr>
<tr>
<td>28.3</td>
<td>Algorithms requirements</td>
<td>901</td>
</tr>
<tr>
<td>28.4</td>
<td>Parallel algorithms</td>
<td>902</td>
</tr>
<tr>
<td>28.5</td>
<td>Non-modifying sequence operations</td>
<td>905</td>
</tr>
<tr>
<td>28.6</td>
<td>Mutating sequence operations</td>
<td>912</td>
</tr>
<tr>
<td>28.7</td>
<td>Sorting and related operations</td>
<td>921</td>
</tr>
<tr>
<td>28.8</td>
<td>C library algorithms</td>
<td>938</td>
</tr>
<tr>
<td>29.1</td>
<td>General</td>
<td>940</td>
</tr>
<tr>
<td>29.2</td>
<td>Definitions</td>
<td>940</td>
</tr>
<tr>
<td>29.3</td>
<td>Numeric type requirements</td>
<td>940</td>
</tr>
<tr>
<td>29.4</td>
<td>The floating-point environment</td>
<td>941</td>
</tr>
<tr>
<td>29.5</td>
<td>Complex numbers</td>
<td>942</td>
</tr>
<tr>
<td>29.6</td>
<td>Random number generation</td>
<td>950</td>
</tr>
<tr>
<td>29.7</td>
<td>Numeric arrays</td>
<td>987</td>
</tr>
<tr>
<td>29.8</td>
<td>Generalized numeric operations</td>
<td>1005</td>
</tr>
<tr>
<td>29.9</td>
<td>Mathematical functions for floating-point types</td>
<td>1016</td>
</tr>
<tr>
<td>30.1</td>
<td>General</td>
<td>1032</td>
</tr>
<tr>
<td>30.2</td>
<td>Iostreams requirements</td>
<td>1032</td>
</tr>
<tr>
<td>30.3</td>
<td>Forward declarations</td>
<td>1033</td>
</tr>
<tr>
<td>30.4</td>
<td>Standard iostream objects</td>
<td>1035</td>
</tr>
<tr>
<td>30.5</td>
<td>Iostreams base classes</td>
<td>1037</td>
</tr>
<tr>
<td>30.6</td>
<td>Stream buffers</td>
<td>1052</td>
</tr>
<tr>
<td>30.7</td>
<td>Formatting and manipulators</td>
<td>1059</td>
</tr>
<tr>
<td>30.8</td>
<td>String-based streams</td>
<td>1082</td>
</tr>
<tr>
<td>30.9</td>
<td>File-based streams</td>
<td>1091</td>
</tr>
<tr>
<td>30.10</td>
<td>Synchronized output streams</td>
<td>1104</td>
</tr>
<tr>
<td>30.11</td>
<td>File systems</td>
<td>1109</td>
</tr>
<tr>
<td>30.12</td>
<td>C library files</td>
<td>1154</td>
</tr>
<tr>
<td>31.1</td>
<td>General</td>
<td>1158</td>
</tr>
<tr>
<td>31.2</td>
<td>Definitions</td>
<td>1158</td>
</tr>
<tr>
<td>31.3</td>
<td>Requirements</td>
<td>1159</td>
</tr>
<tr>
<td>31.4</td>
<td>Header <code>&lt;regex&gt;</code> synopsis</td>
<td>1160</td>
</tr>
<tr>
<td>31.5</td>
<td>Namespace <code>std::regex_constants</code></td>
<td>1166</td>
</tr>
<tr>
<td>31.6</td>
<td>Class <code>regex_error</code></td>
<td>1169</td>
</tr>
<tr>
<td>31.7</td>
<td>Class template <code>regex_traits</code></td>
<td>1169</td>
</tr>
<tr>
<td>31.8</td>
<td>Class template <code>basic_regex</code></td>
<td>1171</td>
</tr>
<tr>
<td>31.9</td>
<td>Class template <code>sub_match</code></td>
<td>1176</td>
</tr>
<tr>
<td>31.10</td>
<td>Class template <code>match_results</code></td>
<td>1181</td>
</tr>
<tr>
<td>31.11</td>
<td>Regular expression algorithms</td>
<td>1185</td>
</tr>
<tr>
<td>31.12</td>
<td>Regular expression iterators</td>
<td>1190</td>
</tr>
<tr>
<td>31.13</td>
<td>Modified ECMAScript regular expression grammar</td>
<td>1195</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>D.14</td>
<td>Deprecated iterator primitives</td>
<td>1320</td>
</tr>
<tr>
<td>D.15</td>
<td>Deprecated <code>shared_ptr</code> observers</td>
<td>1320</td>
</tr>
<tr>
<td>D.16</td>
<td>Deprecated <code>shared_ptr</code> atomic access</td>
<td>1320</td>
</tr>
<tr>
<td>D.17</td>
<td>Deprecated standard code conversion facets</td>
<td>1322</td>
</tr>
<tr>
<td>D.18</td>
<td>Deprecated convenience conversion interfaces</td>
<td>1324</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>1328</td>
</tr>
<tr>
<td>Cross references</td>
<td></td>
<td>1329</td>
</tr>
<tr>
<td>Cross references from ISO C++ 2017</td>
<td></td>
<td>1348</td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>1349</td>
</tr>
<tr>
<td>Index of grammar productions</td>
<td></td>
<td>1379</td>
</tr>
<tr>
<td>Index of library headers</td>
<td></td>
<td>1382</td>
</tr>
<tr>
<td>Index of library names</td>
<td></td>
<td>1384</td>
</tr>
<tr>
<td>Index of implementation-defined behavior</td>
<td></td>
<td>1437</td>
</tr>
</tbody>
</table>
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). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.
2 Normative references

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

(1.2) — ISO/IEC 2382 (all parts), *Information technology — Vocabulary*
(1.3) — ISO/IEC 9899:2011, *Programming languages — C*
(1.4) — ISO/IEC 9945:2003, *Information Technology — Portable Operating System Interface (POSIX)*
(1.7) — ISO 80000-2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*

2 The library described in Clause 7 of ISO/IEC 9899:2011 is hereinafter called the *C standard library.*

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

4 The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called *ECMA-262.*
3 Terms and definitions

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.

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

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

20.3 defines additional terms that are used only in Clause 20 through Clause 33 and Annex D.

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

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

3.2 argument
(function call expression) expression in the comma-separated list bounded by the parentheses (8.5.1.2)

3.3 argument
(function-like macro) sequence of preprocessing tokens in the comma-separated list bounded by the parentheses (19.3)

3.4 argument
(throw expression) operand of throw (8.5.17)

3.5 argument
(template instantiation) constant-expression, type-id, or id-expression in the comma-separated list bounded by the angle brackets (17.3)

3.6 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
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
message belonging to an implementation-defined subset of the implementation’s output messages

3.9 dynamic type
(glvalue) type of the most derived object (6.6.2) to which the glvalue refers

[Example: If a pointer (11.3.1) p whose static type is “pointer to class B” is pointing to an object of class D, derived from B (Clause 13), the dynamic type of the expression *p is ”D”. References (11.3.2) are treated similarly. — end example]
3.10 dynamic type
(prvalue) static type of the prvalue expression

3.11 ill-formed program
program that is not well-formed (3.29)

3.12 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 implementation limits
restrictions imposed upon programs by the implementation

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

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

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

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

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

3.19 signature
(function) name, parameter type list (11.3.5), enclosing namespace (if any), and trailing requires-clause (Clause
11) (if any)

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

3.20 signature
(function template) name, parameter type list (11.3.5), enclosing namespace (if any), return type, template-
head, and trailing requires-clause (Clause 11) (if any)

3.21 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 (11.3.5), class of which the function is a member, 
access-specifiers (if any), ref-qualifier (if any), and trailing requires-clause (Clause 11) (if any)

3.23 signature
(class member function template) name, parameter type list (11.3.5), class of which the function is a 
member, access-specifiers (if any), ref-qualifier (if any), return type (if any), template-head, and trailing requires-
clause (Clause 11) (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 (8.6). — 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

4.1 Implementation compliance

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”.

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:

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[2] that program.

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.

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 17.9.2. —end note]

For classes and class templates, the library Clauses specify partial definitions. Private members (Clause 14) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

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

The names defined in the library have namespace scope (10.3). A C++ translation unit (5.2) obtains access to these names by including the appropriate standard library header (19.2).

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).

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 (20.5.1.3).

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.

Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.3

4.1.1 Abstract machine

The semantic descriptions in this document define a parameterized nondeterministic abstract machine. This document places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.4

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

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

Certain other aspects and operations of the abstract machine are described in this document as unspecified (for example, evaluation of expressions in a \texttt{new-initializer} if the allocation function fails to allocate memory (8.5.2.4)). 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 \texttt{const} object). \footnote{Overloaded operators are never assumed to be associative or commutative.} [\textit{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:

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

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

[\textit{Note}: Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.]\footnote{6} For example, in the following fragment

\begin{verbatim}
int a, b;
/\* ... */
  a = a + 32760 + b + 5;
\end{verbatim}

the expression statement behaves exactly the same as

\[
a = (((a + 32760) + b) + 5);
\]

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 \texttt{int} is \([-32768, +32767]\), the implementation cannot rewrite this expression as

\[
a = ((a + b) + 32765);
\]

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 either as

\[
a = (((a + 32765) + b);
\]

or

\[
a = (a + (b + 32765));
\]

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]
4.2 Structure of this document

Clause 5 through Clause 19 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 21 through Clause 33 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 20.

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_{opt} \}

indicates an optional expression enclosed in braces.

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.

ECMA Script® is a registered trademark of Ecma International.

All rights in these originals are reserved.
5 Lexical conventions [lex]

5.1 Separate translation [lex.separate]

The text of the program is kept in units called source files in this document. A source file together with all the headers (20.5.1.2) and source files included (19.2) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (19.1) preprocessing directives, is called a translation unit. [Note: A C++ program need not all be translated at the same time. — end note]

[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 linkage, manipulation of objects whose identifiers have external 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 [lex.phases]

The precedence among the syntax rules of translation is specified by the following phases.7

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.8 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 (19.3.3), 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.9

7) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.
8) 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.
9) An implementation need not convert all non-corresponding source characters to the same execution character.
6. Adjacent string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token (5.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. \[\text{Note:}\] The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (17.2). \[\text{—end note}\] \[\text{Note:}\] 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. \[\text{—end note}\]

8. Translated translation units and instantiation units are combined as follows: \[\text{Note:}\] Some or all of these may be supplied from a library. \[\text{—end note}\] Each translated translation unit is examined to produce a list of required instantiations. \[\text{Note:}\] This may include instantiations which have been explicitly requested (17.8.2). \[\text{—end note}\] 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. \[\text{Note:}\] An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. \[\text{—end note}\] All the required instantiations are performed to produce instantiation units. \[\text{Note:}\] These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. \[\text{—end note}\] The program is ill-formed if any instantiation fails.

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

### 5.3 Character sets

1. The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:\[10\]

   \begin{verbatim}
   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
   \_ \{ \} [ ] \# \( \) \< \> \, \. \? \* \+ \- / \& \| \( \sim \)
   \end{verbatim}

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

   - hex-quad:
     - \text{hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit}
   - universal-character-name:
     - \text{"u hex-quad}
     - \text{\U hex-quad hex-quad}

   The character designated by the universal-character-name \text{\UNNNNNNNN} is that character whose character short name in ISO/IEC 10646 is \text{\UNNNNNNNN}; the character designated by the universal-character-name \text{"uNNNNN} is that character whose character short name in ISO/IEC 10646 is \text{0000NNNN}. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDBFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for 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 (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.\[11\]

3. The basic execution character set and the basic execution wide-character set shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a null character (respectively, null wide character), whose value is 0. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. In both the source and execution basic character sets, the value of each character after \text{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

---

\[10\] 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.

\[11\] A sequence of characters resembling a universal-character-name in an r-char-sequence (5.13.5) does not form a universal-character-name.
execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

5.4 Preprocessing tokens

preprocessing-token:
  header-name
  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

1 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.

2 A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, 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 19, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

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

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

encoding-prefix opt R raw-string

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

(3.3) Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a header-name (5.8) is only formed within a #include directive (19.2).

[ 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 ]

5 [ 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 [lex.digraph]

Alternative token representations are provided for some operators and punctuators.\(^{12}\)

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.\(^{13}\) The set of alternative tokens is defined in Table 1.

<table>
<thead>
<tr>
<th>Alternative</th>
<th>Primary</th>
<th>Alternative</th>
<th>Primary</th>
<th>Alternative</th>
<th>Primary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;%{</code></td>
<td>and</td>
<td><code>%&gt;</code></td>
<td>bitor</td>
<td><code>&lt;:</code></td>
<td>xor</td>
</tr>
<tr>
<td><code>%&gt;</code></td>
<td>bitor</td>
<td>`</td>
<td>`</td>
<td>`</td>
<td>=`</td>
</tr>
<tr>
<td><code>:&lt;</code></td>
<td>or</td>
<td>`</td>
<td>=`</td>
<td><code>^=</code></td>
<td><code>#:</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>xor</td>
<td><code>-</code></td>
<td><code>not</code></td>
<td><code>:%:</code></td>
<td><code>bitand</code></td>
</tr>
<tr>
<td><code>:%:</code></td>
<td><code>##</code></td>
<td><code>bitand</code></td>
<td><code>&amp;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.6 Tokens [lex.token]

token:

- identifier
- keyword
- literal
- operator
- punctuator

There are five kinds of tokens: identifiers, keywords, literals,\(^{14}\) 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 [lex.comment]

The characters /* start a comment, which terminates with the characters */. These comments do not nest. The characters // start a comment, which terminates immediately before the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only 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 [lex.header]

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 "

\(^{12}\) 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”.

\(^{13}\) Thus the “stringized” values (19.3.2) of [ and `<`: will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

\(^{14}\) Literals include strings and character and numeric literals.

§ 5.8
[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 19.2.

1. 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.\(^{15}\)

### 5.9 Preprocessing numbers

```
pp-number: 
  digit
  . digit
  pp-number digit
  pp-number identifier-nondigit
  pp-number + digit
  pp-number - digit
  pp-number * nondigit
  pp-number % sign
  pp-number E 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

```
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 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.\(^{16}\)

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.

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

\(^{15}\) 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.

\(^{16}\) 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 \`\` in a universal-character-name. Extended characters may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.
Table 2 — Ranges of characters allowed

| 00A8 | 00AA | 00AD | 00AF | 00B2-00B5 |
| 00B7-00BA | 00BC-00BE | 00C0-00D6 | 00D8-00F6 | 00F8-00FF |
| 0100-167F | 1681-180D | 180F-1FFF |
| 200B-200D | 202A-202E | 203F-2040 | 2054 | 2060-206F |
| 2070-218F | 2460-24FF | 2776-2793 | 2C00-2DFF | 2E80-2FFF |
| 300B-300D | 302A-302E | 303F-3040 | 3040 | 3041-D7FF |
| 2192-219E | 2460-24FF | 2776-2793 | 2C00-2DFF | 2E80-2FFF |
| 3004-3007 | 3021-302F | 3031-303D | 303D | 303E-D7FF |
| 200B-200D | 202A-202E | 203F-2040 | 2040 | 2041-D7FF |
| 2192-219E | 2460-24FF | 2776-2793 | 2C00-2DFF | 2E80-2FFF |
| 3004-3007 | 3021-302F | 3031-303D | 303D | 303E-D7FF |

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

| 0300-036F | 1DC0-1DFF | 20D0-20FF | FE20-FE2F |

Table 4 — Identifiers with special meaning

| override | final |

(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 (10.6.1):

Table 5 — Keywords

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

[Note: The export and register keywords are unused but are reserved for future use. — end note]

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

Table 6 — Alternative representations

| and | and_eq | bitand | bitor | compl | not |
| not_eq | or | or_eq | xor | xor_eq |

§ 5.11
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:

```
{ } [ ] # ## ( ) <: : > <% > %% : % : ; : ... 
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).

5.13 Literals

5.13.1 Kinds of literals

There are several kinds of literals.\(^\text{17}\)

```
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\_opt
  octal-literal integer-suffix\_opt
  decimal-literal integer-suffix\_opt
  hexadecimal-literal integer-suffix\_opt

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

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

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

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

binary-digit:
  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
```

\(^{17}\) The term “literal” generally designates, in this document, those tokens that are called “constants” in ISO C.
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, 0x1000000, 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>
<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>long int</td>
</tr>
<tr>
<td></td>
<td>unsigned long long int</td>
<td>long long int</td>
</tr>
<tr>
<td>1 or L</td>
<td>long int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td></td>
<td>long long int</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>Both u or U and 1 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>unsigned long int</td>
</tr>
<tr>
<td>Both u or U and 1l or LL</td>
<td>unsigned long long int</td>
<td>unsigned long long int</td>
</tr>
</tbody>
</table>

18) The digits 8 and 9 are not octal digits.
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

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.

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.

A character literal that begins with u8, such as u8’w’, is a character literal of type char, known as a UTF-8 character literal. The value of a UTF-8 character literal is equal to its ISO 10646 code point value, provided that the code point value is representable with a single UTF-8 code unit (that is, provided it is in the C0 Controls and Basic Latin Unicode block). 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.

A character literal that begins with the letter u, such as u’x’, is a character literal of type char16_t. The value of a char16_t character literal containing a single c-char is equal to its ISO 10646 code point value, provided that the code point value is representable with a single 16-bit code unit (that is, provided it is in the basic multi-lingual plane). If the value is not representable with a single 16-bit code unit, the program is ill-formed. A char16_t character literal containing multiple c-chars is ill-formed.

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

A character literal that begins with the letter u, such as u’x’, is a character literal of type char16_t. The value of a char16_t character literal containing a single c-char is equal to its ISO 10646 code point value, provided that the code point value is representable with a single 16-bit code unit (that is, provided it is in the basic multi-lingual plane). If the value is not representable with a single 16-bit code unit, the program is ill-formed. A char16_t character literal containing multiple c-chars is ill-formed.
A character literal that begins with the letter \texttt{L}, such as \texttt{L'z'}, is a \textit{wide-character literal}. A wide-character literal has type \texttt{wchar_t}.\footnote{They are intended for character sets where a character does not fit into a single byte.} The value of a wide-character literal containing a single \texttt{c-char} has value equal to the numerical value of the encoding of the \texttt{c-char} in the execution wide-character set, unless the \texttt{c-char} has no representation in the execution wide-character set, in which case the value is implementation-defined. \[ Note: \] The type \texttt{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 \texttt{c-chars} is implementation-defined.

Certain non-graphic characters, the single quote ‘, the double quote “, the question mark ?,\footnote{Using an escape sequence for a question mark is supported for compatibility with ISO C++ 2014 and ISO C.} and the backslash \texttt{\}, can be represented according to Table 8. The double quote “ and the question mark ?, can be represented as themselves or by the escape sequences \texttt{\"} and \texttt{\?} respectively, but the single quote ‘ and the backslash \texttt{\} shall be represented by the escape sequences \texttt{\'} and \texttt{\\} 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.

\begin{center}
\begin{tabular}{|l|l|}
\hline
new-line & NL[LF] \texttt{\n} \\
horizontal tab & HT \texttt{\t} \\
vertical tab & VT \texttt{\v} \\
backspace & BS \texttt{\b} \\
carriage return & CR \texttt{\r} \\
form feed & FF \texttt{\f} \\
alert & BEL \texttt{\a} \\
backslash & \texttt{\} \\
question mark & ? \texttt{\?} \\
single quote & ‘ \texttt{\'} \\
double quote & “ \texttt{\"} \\
octal number & \texttt{000} \texttt{\ooo} \\
hex number & \texttt{hhh} \texttt{\xhhh} \\
\hline
\end{tabular}
\end{center}

The escape \texttt{\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 \texttt{\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 \texttt{char} (for character literals with no prefix) or \texttt{wchar_t} (for character literals prefixed by \texttt{L}). \[ Note: \] If the value of a character literal prefixed by \texttt{u}, \texttt{u8}, or \texttt{U} is outside the range defined for its type, the program is ill-formed. \[— end note\]

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

\section*{5.13.4 Floating literals} \[lex.fcon\]

\begin{verbatim}
float-literal:
  decimal-float-literal
  hexadecimal-float-literal

decimal-float-literal:
  fractional-constant exponent-part opt floating-suffix opt
  digit-sequence exponent-part floating-suffix opt

hexadecimal-float-literal:
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffix opt
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-suffix opt
\end{verbatim}
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 l 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 not 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 [lex.string]

stringLiteral:
  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.

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\\nb\nc") == 0);

   — end note ]

5 [ Example: The raw string

        R"a(\n a" is equivalent to "\n)a. 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, 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.

8 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type “array of \n const char”, where n is the size of the string as defined below, and has static storage duration (6.6.4).

9 For a UTF-8 string literal, 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.

10 A string-literal that begins with u, such as u"asdf", is a char16_t string literal. A char16_t string literal has type “array of \n const char16_t”, where n is the size of the string as defined below; it is initialized with the given characters. A single c-char may produce more than one char16_t character in the form of surrogate pairs.

11 A string-literal that begins with U, such as U"asdf", is a char32_t string literal. A char32_t string literal has type “array of \n const char32_t”, where n is the size of the string as defined below; it is initialized with the given characters.

12 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 9 — String literal concatenations

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

Characters in concatenated strings are kept distinct.

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

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

Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (5.13.3), except that the single quote ' is representable either by itself or by the escape sequence \', and the double quote " shall be preceded by a \, and except that a universal-character-name in a char16_t string literal may yield a surrogate pair. In a narrow string literal, a universal-character-name may map to more than one char element due to multibyte encoding. The size of a char32_t or wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating U'\0' or L'\0'. The size of a char16_t string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating u'\0'. [Note: The size of a char16_t string literal is the number of code units, not the number of characters. —end note] Within char32_t and char16_t string literals, any universal-character-names shall be within the range 0x0 to 0x10FFFF. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating \0'.

Evaluating a string-literal results in a string literal object with static storage duration, initialized from the given characters as specified above. Whether all string literals are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a string-literal yield the same or a different object is unspecified. [Note: The effect of attempting to modify a string literal is undefined. —end note]

5.13.6 Boolean literals

Boolean literals:

**false**

**true**

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

5.13.7 Pointer literals

Pointer literals:

nullptr

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 7.11 and 7.12. —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 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

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 (16.5.8). 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

  operator "" X(nULL)

Otherwise, S shall contain a raw literal operator or a literal operator template (16.5.8) but not both. If S contains a raw literal operator, the literal L is treated as a call of the form

  operator "" X("n")

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

  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]

4 If L is a user-defined-floating-literal, let f be the literal without its ud-suffix. If S contains a literal operator with parameter type long double, the literal L is treated as a call of the form

  operator "" X(fL)

Otherwise, S shall contain a raw literal operator or a literal operator template (16.5.8) but not both. If S contains a raw literal operator, the literal L is treated as a call of the form

  operator "" X("f")

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

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

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

5 If L is a user-defined-string-literal, let str be the literal without its ud-suffix and let len be the number of code units in str (i.e., its length excluding the terminating null character). The literal L is treated as a call of the form
operator "" \( X(\text{str}, \text{len}) \)

6 If \( L \) is a user-defined-character-literal, let \( ch \) be the literal without its ud-suffix. \( S \) shall contain a literal operator (16.5.8) whose only parameter has the type of \( ch \) and the literal \( L \) is treated as a call of the form

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

7 [Example:

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

—end example]

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

9 [Example:

\[
\begin{align*}
\text{int main()} \{ \\
\quad \text{L""A"" } \text{""B"" } \text{""C""}_x; & \quad \text{// OK: same as L""ABC""}_x \\
\quad \text{""P""}_x \text{""Q"" } \text{""R""}_y; & \quad \text{// error: two different ud-suffixes}
\end{align*}
\]

—end example]
6 Basic concepts

[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 parameter pack.

A name is a use of an identifier (5.10), operator-function-id (16.5), literal-operator-id (16.5.8), conversion-function-id (15.3.2), or template-id (17.2) that denotes an entity or label (9.6.4, 9.1).

Every name that denotes an entity is introduced by a declaration. Every name that denotes a label is introduced either by a goto statement (9.6.4) or a labeled-statement (9.1).

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 (6.6.4.3), a structured binding (11.5) whose corresponding variable is such an entity, or the *this object (8.4.2).

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

1. they are identifiers composed of the same character sequence, or
2. they are operator-function-ids formed with the same operator, or
3. they are conversion-function-ids formed with the same type, or
4. they are template-ids that refer to the same class, function, or variable (17.5), or
5. they are the names of literal operators (16.5.8) 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 10) 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 attributes of these names. A declaration may also have effects including:

1. a static assertion (Clause 10),
2. controlling template instantiation (17.8.2),
3. guiding template argument deduction for constructors (17.10),
4. use of attributes (Clause 10), and
5. nothing (in the case of an empty-declaration).

A declaration is a definition unless

1. it declares a function without specifying the function’s body (11.4),
2. it contains the extern specifier (10.1.1) or a linkage-specification (10.5) and neither an initializer nor a function-body.

[21] Appearing inside the brace-enclosed declaration-seq in a linkage-specification does not affect whether a declaration is a definition.
(2.3) — it declares a non-inline static data member in a class definition (12.2, 12.2.3),
(2.4) — 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.1),
(2.5) — it is a class name declaration (12.1),
(2.6) — it is an opaque-enum-declaration (10.2),
(2.7) — it is a template-parameter (17.1),
(2.8) — it is a parameter-declaration (11.3.5) in a function declarator that is not the declarator of a function-definition,
(2.9) — it is a typedef declaration (10.1.3),
(2.10) — it is an alias-declaration (10.1.3),
(2.11) — it is a using-declaration (10.3.3),
(2.12) — it is a deduction-guide (17.10),
(2.13) — it is a static_assert-declaration (Clause 10),
(2.14) — it is an attribute-declaration (Clause 10),
(2.15) — it is an empty-declaration (Clause 10),
(2.16) — it is a using-directive (10.3.4),
(2.17) — it is an explicit instantiation declaration (17.8.2), or
(2.18) — it is an explicit specialization (17.8.3) whose declaration is not a definition.

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

whereas these are just declarations:

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

/* end example */
```

3 [Note: In some circumstances, C++ implementations implicitly define the default constructor (15.1), copy
constructor (15.8), move constructor (15.8), copy assignment operator (15.8), move assignment operator (15.8),
or destructor (15.4) member functions. —end note] [Example: Given

```c
#include <string>

struct C {
  std::string s; // std::string is the standard library class (Clause 24)
};

int main() {
  C a;

```
```c++
C b = a;
b = a;
}
```

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

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

—end example

4 [ Note: A class name can also be implicitly declared by an elaborated-type-specifier (10.1.7.3). — end note ]

5 A program is ill-formed if the definition of any object gives the object an incomplete type (6.7).

### 6.2 One-definition rule

1 No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.

2 An expression is potentially evaluated unless it is an unevaluated operand (8.2) or a subexpression thereof.

The set of potential results of an expression e is defined as follows:

1. If e is an id-expression (8.4.4), the set contains only e.
2. If e is a subscripting operation (8.5.1.1) with an array operand, the set contains the potential results of that operand.
3. If e is a class member access expression (8.5.1.5), the set contains the potential results of the object expression.
4. If e is a pointer-to-member expression (8.5.4) whose second operand is a constant expression, the set contains the potential results of the object expression.
5. If e has the form (e1), the set contains the potential results of e1.
6. If e is a glvalue conditional expression (8.5.16), the set is the union of the sets of potential results of the second and third operands.
7. If e is a comma expression (8.5.19), the set contains the potential results of the right operand.
8. Otherwise, the set is empty.

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

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

```c++
struct S { static const int x = 0; }
const int &f(const int &r);
int n = b ? (1, S::x) // S::x is not odr-used here
    : f(S::x); // S::x is odr-used here, so a definition is required

— end example] — end note ]

3 A function is named by an expression as follows:

1. A function whose name appears in an expression is named by that expression if it is the unique lookup result or the selected member of a set of overloaded functions (6.4, 16.3, 16.4), unless it is a pure virtual function and either its name is not explicitly qualified or the expression forms a pointer to member (8.5.2.1). [ Note: This covers taking the address of functions (7.3, 8.5.2.1), calls to named functions (8.5.1.2), operator overloading (Clause 16), user-defined conversions (15.3.2), allocation functions for placement new-expressions (8.5.2.4), as well as non-default initialization (11.6). A constructor selected to copy or move an object of class type is considered to be named by an expression even if the call is actually elided by the implementation (15.8). — end note ]
An allocation or deallocation function for a class is named by a new-expression as specified in 8.5.2.4 and 15.5.

A deallocation function for a class is named by a delete expression as specified in 8.5.2.5 and 15.5.

A variable \(x\) whose name appears as a potentially-evaluated expression \(ex\) is odr-used by \(ex\) unless applying the lvalue-to-rvalue conversion (7.1) to \(x\) yields a constant expression (8.6) that does not invoke any non-trivial functions and, if \(x\) is an object, \(ex\) is an element of the set of potential results of an expression \(e\), where either the lvalue-to-rvalue conversion (7.1) is applied to \(e\), or \(e\) is a discarded-value expression (8.2).

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

\(*\text{this}\) is odr-used if \(\text{this}\) appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (12.2.2)).

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. 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 (15.4).²²

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 15.8. A constructor for a class is odr-used as specified in 11.6. A destructor for a class is odr-used if it is potentially invoked (15.4).

A local entity (Clause 6) is odr-usable in a declarative region (6.3.1) if:

1. The local entity is either not \(*\text{this}\), 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
2. For each intervening declarative region (6.3.1) between the point at which the entity is introduced and the region (where \(*\text{this}\) is considered to be introduced within the innermost enclosing class or non-lambda function definition scope), either:
   1. The declarative region is a block scope, or
   2. The declarative region is the function parameter scope of a lambda-expression that has a simple-capture naming the entity or has a capture-default.

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

Example:

```cpp
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
    [&] { [n] { return n; }; } // OK
}
```

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 (9.4.1); 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 15.1, 15.4 and 15.8). An inline function or variable shall be defined in every translation unit in which it is odr-used outside of a discarded statement.

Exactly one definition of a class is required in a translation unit if 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; // declare X as a struct type
struct X* x1; // use X in pointer formation
X* x2; // use X in pointer formation
```

²² An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.
— end example] [ Note: The rules for declarations and expressions describe in which contexts complete class types are required. A class type \( T \) must be complete if:

- an object of type \( T \) is defined (6.1), or
- a non-static class data member of type \( T \) is declared (12.2), or
- \( T \) is used as the allocated type or array element type in a new-expression (8.5.2.4), or
- an lvalue-to-rvalue conversion is applied to a glvalue referring to an object of type \( T \) (7.1), or
- an expression is converted (either implicitly or explicitly) to type \( T \) (Clause 7, 8.5.1.3, 8.5.1.7, 8.5.1.9, 8.5.3), or
- an expression that is not a null pointer constant, and has type other than \( cv \) void\*, is converted to the type pointer to \( T \) or reference to \( T \) using a standard conversion (Clause 7), a dynamic_cast (8.5.1.7) or a static_cast (8.5.1.9), or
- a class member access operator is applied to an expression of type \( T \) (8.5.1.5), or
- the typeid operator (8.5.1.8) or the sizeof operator (8.5.2.3) is applied to an operand of type \( T \), or
- a function with a return type or argument type of type \( T \) is defined (6.1) or called (8.5.1.2), or
- a class with a base class of type \( T \) is defined (Clause 13), or
- an lvalue of type \( T \) is assigned to (8.5.18), or
- the type \( T \) is the subject of an alignof expression (8.5.2.6), or
- an exception-declaration has type \( T \), reference to \( T \), or pointer to \( T \) (18.3).

— end note]

There can be more than one definition of a class type (Clause 12), enumeration type (10.2), inline function with external linkage (10.1.6), inline variable with external linkage (10.1.6), class template (Clause 17), non-static function template (17.6.6), concept (17.6.8), static data member of a class template (17.6.1.3), member function of a class template (17.6.1.1), or template specialization for which some template parameters are not specified (17.8, 17.6.5) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named \( D \) defined in more than one translation unit, then

- each definition of \( D \) shall consist of the same sequence of tokens; and
- 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 (16.3) and after matching of partial template specialization (17.9.3), except that a name can refer to

- a non-volatile const object with internal or no linkage if the object
  - has the same literal type in all definitions of \( D \),
  - is initialized with a constant expression (8.6),
  - is not odr-used in any definition of \( D \), and
  - has the same value in all definitions of \( D \),
  or
- a reference with internal or no linkage initialized with a constant expression such that the reference refers to the same entity in all definitions of \( D \);

and

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

23) 11.3.6 describes how default argument names are looked up.

§ 6.2 28
(12.6) if D is a class with an implicitly-declared constructor (15.1), 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:

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

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

—end example]

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 (17.7.3), and also to dependent names at the point of instantiation (17.7.2). If the definitions of D satisfy all these requirements, 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 (8.4.5) 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 [basic.scope]

6.3.1 Declarative regions and scopes [basic.scope.declarative]

1 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 discontinuous 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.

2 [Example: In

```cpp
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 (14.3), certain uses of the elaborated-type-specifier (10.1.7.3), and using-directives (10.3.4) alter this general behavior.

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

1. they shall all refer to the same entity, or all refer to functions and function templates; or
2. exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same variable, non-static data member, or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (6.3.10). [Note: A namespace name or a class template name must be unique in its declarative region (10.3.2, Clause 17). —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 (10.1.7.3) and friend declarations (14.3) 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 [basic.scope.pdecl]

1. The point of declaration for a name is immediately after its complete declarator (Clause 11) and before its initializer (if any), except as noted below. [Example:

   ```
   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:

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

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

2. 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 12). The point of declaration for an enumeration is immediately after the identifier (if any) in either its enum-specifier (10.2) or its first opaque-enum-declaration (10.2), whichever comes first. The point of declaration of an alias or alias template immediately follows the type-id to which the alias refers.

3. The point of declaration of a using-declarator that does not name a constructor is immediately after the using-declarator (10.3.3).

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

   ```
   const int x = 12;
   { enum { x = x }; }
   ```

   Here, the enumerator x is initialized with the value of the constant x, namely 12. —end example]

5. 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,

   ```
   struct X {
   enum E { z = 16 };
   int b[X::z]; // OK
   };
   ```

   —end note]

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

   ```
   struct X {
   enum E { z = 16 };
   int b[X::z]; // OK
   };
   ```

   — for a declaration of the form

   ```
   struct X {
   enum E { z = 16 };
   int b[X::z]; // OK
   };
   ```

   — for a declaration of the form
(7.2) the identifier is declared to be a class-name in the scope that contains the declaration, otherwise

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 10.1.7.3. —end note]

6.3.3 Block scope [basic.scope.block]

1 A name declared in a block (9.3) 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.

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

3 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 9.4.

6.3.4 Function parameter scope [basic.scope.param]

1 A function parameter (including one appearing in a lambda-declarator) or function-local predefined variable (11.4) 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 [basic.funscope]

1 Labels (9.1) 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 (10.3.4) that nominates the member’s namespace, the member’s potential scope includes that portion of the potential scope of the using-directive that follows the member’s point of declaration. [Example:

```c
namespace N {
  int i;
  int g(int a) { return a; }
  int j();
  void q();
}
namespace { int l=1; }
// the potential scope of l is from its point of declaration to the end of the translation unit
namespace N {
  int g(char a) { // overloads N::g(int)
    return l+a; // l is from unnamed namespace
  }
  int i; // error: duplicate definition
  int j(); // OK: duplicate function declaration
  int j() {
    return g(i); // calls N::g(int)
  }
  int q(); // error: different return type
}
```
—end example]

2 A namespace member can also be referred to after the :: scope resolution operator (8.4) 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.

3 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 function bodies, default arguments, noexcept-specifiers, and brace-or-equal-initializers of non-static data members in that class (including such things in nested classes).

2 A name 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 (11.3.6)).

[Example:

typedef int c;
enum { i = 1 };
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 (Clause 13) from its class,
- after the . operator applied to an expression of the type of its class (8.5.1.5) or a class derived from its class,
- after the -> operator applied to a pointer to an object of its class (8.5.1.5) or a class derived from its class,
- after the :: scope resolution operator (8.4) applied to the name of its class or a class derived from its class.

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

(6.1) — in the scope of its class (as described above) or a class derived (Clause 13) from its class,
(6.2) — after the . operator applied to an expression of the type of its class (8.5.1.5) or a class derived from its class,
(6.3) — after the -> operator applied to a pointer to an object of its class (8.5.1.5) or a class derived from its class,
(6.4) — after the :: scope resolution operator (8.4) applied to the name of its class or a class derived from its class.

6.3.8 Enumeration scope

1 The name of a scoped enumerator (10.2) 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

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

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

```c
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]

3 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,

```c
template<class T, T* p, class U = T> class X { /* ... */};
template<class T> void f(T* p = new T);
```
This also implies that a template-parameter can be used in the specification of base classes. For example,

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

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

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

4. [ Note: Because the name of a template parameter cannot be redeclared within its potential scope (17.7.1), a template parameter’s scope is often its potential scope. However, it is still possible for a template parameter name to be hidden; see 17.7.1. ] — end note

### 6.3.10 Name hiding

A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class (13.2).

A class name (12.1) or enumeration name (10.2) can be hidden by the name of a variable, data member, function, or enumerator declared in the same scope. If a class or enumeration name and a variable, data member, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the variable, data member, function, or enumerator name is visible.

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 (Clause 13) hides the declaration of a member of a base class of the same name; see 13.2.

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.

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 (10.1.3), namespace-names (10.3), and class-names (12.1)) 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 declare the same entity or shall all declare functions; in the latter case, the declarations are said to form a set of overloaded functions (16.1). Overload resolution (16.3) takes place after name lookup has succeeded. The access rules (Clause 14) 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 attributes introduced by the name’s declaration used further in expression processing (Clause 8).

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

The injected-class-name of a class (Clause 12) is also considered to be a member of that class for the purposes of name hiding and lookup.

[ 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.

The declarations from the namespace nominated by a using-directive become visible in a namespace enclosing the using-directive; see 10.3.4. 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.

3 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 17.2). For example,

```cpp
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);
}
```

```cpp
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,

```cpp
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—

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

5 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.

6 In the definition of a function that is a member of namespace N, a name used after the function’s declarator-id2A shall be declared before its use in the block in which it is used or in one of its enclosing blocks (9.3) 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:

```cpp
namespace A {
    namespace N {
        void f();
    }
}
```

```cpp
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—

24) 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.

§ 6.4.1
A name used in the definition of a class \( X \) outside of a member function body, default argument, \textit{noexcept-specifier}, \textit{brace-or-equal-initializer} of a non-static data member, or nested class definition\(^{25}\) shall be declared in one of the following ways:

\begin{enumerate}
\item[(7.1)] before its use in class \( X \) or be a member of a base class of \( X \) (13.2), or
\item[(7.2)] if \( X \) is a nested class of class \( Y \) (12.2.5), before the definition of \( X \) in \( Y \), or shall be a member of a base class of \( Y \) (this lookup applies in turn to \( Y \)'s enclosing classes, starting with the innermost enclosing class),\(^{26}\) or
\item[(7.3)] if \( X \) is a local class (12.4) 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
\item[(7.4)] if \( X \) is a member of namespace \( N \), or is a nested class of a class that is a member of \( N \), or is a local class or a nested class within a local class of a function that is a member of \( N \), before the definition of class \( X \) in namespace \( N \) or in one of \( N \)'s enclosing namespaces.
\end{enumerate}

\[\text{Example:}\]
\begin{verbatim}
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{verbatim}

\[\text{—end example}\]  
\[\text{[Note: When looking for a prior declaration of a class or function introduced by a friend declaration, scopes outside of the innermost enclosing namespace scope are not considered; see 10.3.1.2. —end note]}\]

\[\text{[Note: 6.3.7 further describes the restrictions on the use of names in a class definition. 12.2.5 further describes the restrictions on the use of names in nested class definitions. 12.4 further describes the restrictions on the use of names in local class definitions. —end note]}\]

For the members of a class \( X \), a name used in a member function body, in a default argument, in a \textit{noexcept-specifier}, in the \textit{brace-or-equal-initializer} of a non-static data member (12.2), or in the definition of a class member outside of the definition of \( X \), following the member’s \textit{declarator-id}\(^{27}\), shall be declared in one of the following ways:

\begin{enumerate}
\item[(8.1)] before its use in the block in which it is used or in an enclosing block (9.3), or
\item[(8.2)] shall be a member of class \( X \) or be a member of a base class of \( X \) (13.2), or
\item[(8.3)] if \( X \) is a nested class of class \( Y \) (12.2.5), 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),\(^{28}\) or
\item[(8.4)] if \( X \) is a local class (12.4) 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
\item[(8.5)] if \( X \) is a member of namespace \( N \), or is a nested class of a class that is a member of \( N \), or is a local class or a nested class within a local class of a function that is a member of \( N \), before the use of the name, in namespace \( N \) or in one of \( N \)'s enclosing namespaces.
\end{enumerate}

\[\text{25) This refers to unqualified names following the class name; such a name may be used in the \textit{base-clause} or may be used in the class definition.}\]

\[\text{26) This lookup applies whether the definition of \( X \) is nested within \( Y \)'s definition or whether \( X \)'s definition appears in a namespace scope enclosing \( Y \)'s definition (12.2.5).}\]

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

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

```cpp
class B { };
namespace M {
    namespace N {
    class X : public B {
        void f();
    };
}
}
void M::N::X::f() {
i = 16;
}
```

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

—end example—

Note: 12.2.1 and 12.2.3 further describe the restrictions on the use of names in member function definitions. 12.2.5 further describes the restrictions on the use of names in the scope of nested classes. 12.4 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 (14.3) 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 (13.2). If it is not found, or if the name is part of a template-argument in the declarator-id, the look up is as described for unqualified names in the definition of the class granting friendship. [Example:

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

—end example—

Note: 11.3.6 further describes the restrictions on the use of names in default arguments. 15.6.2 further describes the restrictions on the use of names in a ctor-initializer. —end note—

11 During the lookup for a name used as a default argument (11.3.6) in a function parameter-declaration-clause or used in the expression of a mem-initializer for a constructor (15.6.2), 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: 11.3.6 further describes the restrictions on the use of names in default arguments. 15.6.2 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 (12.2.3.2) (after the qualified-id of the static member) is looked up as if the name was used in a member function of X. [Note: 12.2.3.2 further describes the restrictions on the use of names in the definition of a static data member. —end note—

§ 6.4.1 37
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 18) 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 17.7. —end note]

### 6.4.2 Argument-dependent name lookup [basic.lookup.argdep]

When the postfix-expression in a function call (8.5.1.2) 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 (14.3) 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:

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

void g() {
    N::S s;
    f(s); // OK: calls N::f
    (f)(s); // error: N::f not considered; parentheses prevent argument-dependent lookup
}
```
—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 classes to be considered. The sets of namespaces and classes 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 classes are determined in the following way:

1. If `T` is a fundamental type, its associated sets of namespaces and classes are both empty.
2. If `T` is a class type (including unions), its associated classes 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 classes. Furthermore, if `T` is a class template specialization, its associated namespaces and classes also include: the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template type parameters); 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. If it is a class member, its associated class is the member’s class; else it has no associated class.
4. If `T` is a pointer to `U` or an array of `U`, its associated namespaces and classes are those associated with `U`.
5. If `T` is a function type, its associated namespaces and classes are those associated with the function parameter types and those associated with the return type.
If \( T \) is a pointer to a member function of a class \( X \), its associated namespaces and classes are those associated with the function parameter types and return type, together with those associated with \( X \).

If an associated namespace is an inline namespace (10.3.1), 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 classes and namespaces are the union of those associated with each of the members of the set, i.e., the classes 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 classes 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

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

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

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

```cpp
class A {
    public:
        static int n;
};
int main() {
    int A;
    A::n = 42;  // OK
```]

When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (6.4.3.2) except that:

- Any using-directives in the associated namespace are ignored.
- Any namespace-scope friend functions or friend function templates (14.3) declared in associated classes are visible within their respective namespaces even if they are not visible during an ordinary lookup (10.3.1.2).
- All names except those of (possibly overloaded) functions and function templates are ignored.

### 6.4.3 Qualified name lookup

The name of a class or namespace member or enumerator can be referred to after the :: scope resolution operator (8.4) applied to a nested-name-specifier that denotes its class, namespace, or enumeration. If a :: scope resolution operator in a nested-name-specifier is not preceded by a decltype-specifier, lookup of the name preceding that :: considers only namespaces, types, and templates whose specializations are types. If the name found does not designate a namespace or a class, enumeration, or dependent type, the program is ill-formed. [Example:

```cpp
class A {
    public:
        static int n;
};
int main() {
    int A;
    A::n = 42;  // OK
```]
A b;  // ill-formed: a does not name a type

— end example]

2 [Note: Multiply qualified names, such as N1::N2::N3::n, can be used to refer to members of nested classes (12.2.5) or members of nested namespaces. — end note]

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

```c
class X { }
class C {
    class X { }
    static const int number = 50;
    static X arr[number];
};
X C::arr[number];  // ill-formed:
    // equivalent to ::X C::arr[C::number];
    // and not to C::X C::arr[C::number];

— end example]

4 A name prefixed by the unary scope operator :: (8.4) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a using-directive (6.4.3.2). The use of :: allows a global name to be referred to even if its identifier has been hidden (6.3.10).

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

6 If a pseudo-destructor-name (8.5.1.4) contains a nested-name-specifier, the type-names are looked up as types in the scope designated by the nested-name-specifier. Similarly, in a qualified-id of the form:

```
nested-name-specifier opt class-name :: - class-name
```

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

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

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

1 If the nested-name-specifier of a qualified-id nominates a class, the name specified after the nested-name-specifier is looked up in the scope of the class (13.2), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (Clause 13). [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) — the lookup for a destructor is as specified in 6.4.3;
(1.2) — a conversion-type-id of a conversion-function-id is looked up in the same manner as a conversion-type-id in a class member access (see 6.4.5):
— the names in a template-argument of a template-id are looked up in the context in which the entire postfix-expression occurs.

— the lookup for a name specified in a using-declaration (10.3.3) also finds class or enumeration names hidden within the same scope (6.3.10).

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

— if the name specified after the nested-name-specifier, when looked up in C, is the injected-class-name of \( C \) (Clause 12), or

— in a using-declarator of a using-declaration (10.3.3) 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:

```cpp
struct A { A(); };  
struct B: public A { B(); }; 
A::A() { }  
B::B() { }  
B::A ba;  // object of type A  
A::A a;  // error, A::A is not a type name  
struct A::A a2;  // object of type A  
```

— end example]

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

### 6.4.3.2 Namespace members [namespace.qual]

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

2 For a namespace \( X \) and name \( m \), the namespace-qualified lookup set \( S(X, m) \) is defined as follows: Let \( S'(X, m) \) be the set of all declarations of \( m \) in \( X \) and the inline namespace set of \( X \) (10.3.1). 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 (10.3.3), \( S(X, m) \) 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

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

[ Example:

namespace A {
    int a;
}

namespace B {
    using namespace A;
}

namespace C {
    using namespace A;
}

namespace BC {
    using namespace B;
    using namespace C;
}

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

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

void g()
{
    BD::a++;  // OK: S is \{A::A::a\}
}

—end example —end note

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

```cpp
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
either introduce 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:

```cpp
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

§ 6.4.3.2
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 (10.3.1) 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);
   }
}
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  [basic.lookup.elab]

1 An elaborated-type-specifier (10.1.7.3) 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).

2 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.

3 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;  // OK: Refers to Node at global scope
   struct Data* Data;  // OK: Declares type Data at global scope and member Data
};

struct Data {
   struct Node* Node;  // OK: Refers to Node at global scope
   friend struct ::Glob;  // error: Glob is not declared, cannot introduce a qualified type (10.1.7.3)
   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 (10.1.7.3)
struct Base::Data; // error: cannot introduce a qualified type (10.1.7.3)
struct Base::Datum; // error: Datum undefined
struct Base::Data* pBase; // OK: refers to nested Data
—end example

6.4.5 Class member access
[basic.lookup.classref]

1 In a class member access expression (8.5.1.5), 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 (17.2) or a less-than operator. The identifier is first looked up in the class of the object expression. 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 (8.5.1.5) is an unqualified-id, and the type of the object expression is of a class type C, the unqualified-id is looked up in the scope of class C. For a pseudo-destructor call (8.5.1.4), the unqualified-id is looked up in the context of the complete postfix-expression. If the type of the object expression is not a class type, the unqualified-id is looked up in the scope of the complete postfix-expression.

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

```cpp
struct A { }

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

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

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

the class-name-or-namespace-name following the . or -> operator is first looked up in the class of the object expression 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
::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 (17.2), 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 and the name, if found, is used. Otherwise it is looked up in the context of the entire postfix-expression. In each of these lookups, only names that denote types or templates whose specializations are types are considered. [ Example:

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

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

—end example—

6.4.6 Using-directives and namespace aliases [basic.lookup.udir]

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 [basic.link]

A program consists of one or more translation units (Clause 5) linked together. A translation unit consists of a sequence of declarations.

translation-unit:
    declaration-seqopt

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:

(2.1) — 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.

(2.2) — When a name has internal linkage, the entity it denotes can be referred to by names from other scopes in the same translation unit.

(2.3) — 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

(3.1) — a variable, function or function template that is explicitly declared static; or,

(3.2) — a non-inline variable of non-volatile const-qualified type that is neither explicitly declared extern nor previously declared to have external linkage; or

(3.3) — a data member of an anonymous union.

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 has the same linkage as the enclosing namespace if it is the name of

(4.1) — a variable; or

(4.2) — a function; or

(4.3) — a named class (Clause 12), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (10.1.3); or

(4.4) — a named enumeration (10.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (10.1.3); or

(4.5) — a template.

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 (10.1.3), 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 there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, 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();
static int i = 0;       // #1
void g() {
    extern void f();   // internal linkage
    int i;             // #2: i has no linkage
    {
        extern void f();   // internal linkage
        extern int i;      // #3: external linkage, ill-formed
        
    }
}
```

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

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

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

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

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

—end example]

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

9 A type is said to have linkage if and only if:

- it is a class or enumeration type that is named (or has a name for linkage purposes (10.1.3)) and the name has linkage; or
- it is an unnamed class or unnamed enumeration that is a member of a class with linkage; or
- it is a specialization of a class template (Clause 17); or
- it is a fundamental type (6.7.1); or
- it is a compound type (6.7.2) other than a class or enumeration, compounded exclusively from types that have linkage; or
- it is a cv-qualified (6.7.3) version of a type that has linkage.

A type without linkage shall not be used as the type of a variable or function with external linkage unless:

- the entity has C language linkage (10.5), or
- the entity is not odr-used (6.2) or is defined in the same translation unit.

[Note: In other words, a type without linkage contains a class or enumeration that cannot be named outside its translation unit. An entity with external linkage declared using such a type could not correspond to any other entity in another translation unit of the program and thus must be defined in the translation unit if it is odr-used. Also note that classes with linkage may contain members whose types do not have linkage, and that typedef names are ignored in the determination of whether a type has linkage. —end note]

30) A class template has the linkage of the innermost enclosing class or namespace in which it is declared.
Example:

```cpp
template <class T> struct B {
    void g(T) { }
    void h(T);
    friend void i(B, T) { }
};
void f() {
    struct A { int x; }; // no linkage
    A a = { 1 };        // declares B<A>::g(A) and B<A>::h(A)
    B<A> ba;            // error: B<A>::h(A) not defined in the translation unit
    ba.g(a);            // OK
    ba.h(a);            // OK
    i(ba, a);           // OK
}
```

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

1. both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and
2. both names refer to members of the same namespace or to members, not by inheritance, of the same class; and
3. when both names denote functions, the parameter-type-lists of the functions (11.3.5) are identical; and
4. when both names denote function templates, the signatures (17.6.6.1) are the same.

After all adjustments of types (during which typedefs (10.1.3) 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 (11.3.4). 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 (10.5).

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.

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 structure declared as

```cpp
struct {
    char a;
```
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 [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is created by a definition (6.1), by a new-expression (8.5.2.4), when implicitly changing the active member of a union (12.3), or when a temporary object is created (7.4, 15.2). An object occupies a region of storage in its period of construction (15.7), throughout its lifetime (6.6.3), and in its period of destruction (15.7). [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 (6.6.4) which influences its lifetime (6.6.3). An object has a type (6.7). Some objects are polymorphic (13.3); the implementation generates information associated with each such object that makes it possible to determine that object’s type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the expressions (8.5) used to access them.

2 Objects can contain other objects, called subobjects. A subobject can be a member subobject (12.2), a base class subobject (Clause 13), 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 (12.3)
   u.f = 5.f;  // OK, creates new subobject of u
   X *p = new (ku.x) X {2};  // OK
   assert(p->n == 2);  // OK
   assert(*std::launder(&u.x.n) == 2);  // undefined behavior, u.x does not name new subobject
}

—end example]

3 If a complete object is created (8.5.2.4) in storage associated with another object e of type “array of N unsigned char” or of type “array of N std::byte” (21.2.1), 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[max(sizeof(T)...)];
};

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

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

— end example ]

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

5 For every object x, there is some object called the complete object of x, determined as follows:
(5.1) — If x is a complete object, then the complete object of x is itself.
(5.2) — Otherwise, the complete object of x is the complete object of the (unique) object that contains x.

6 If a complete object, a data member (12.2), 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 Unless it is a bit-field (12.2.4), a most derived object shall have a nonzero size and shall occupy one or more bytes of storage. Base class subobjects may have zero size. An object of trivially copyable or standard-layout type (6.7) shall occupy contiguous bytes of storage.

8 Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two objects a and b 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 base class subobject of zero size and they are of different types; otherwise, they have distinct addresses.32 [ Example:
static const char test1 = 'x';
static const char test2 = 'x';
const bool b = &test1 != &test2; // always true
— end example ]

9 [ 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 lifetime [basic.life]

1 The lifetime of an object or reference is a runtime property of the object or reference. An object is said to have non-vacuous initialization if it is of a class or aggregate type and it or one of its subobjects is initialized by a constructor other than a trivial default constructor. [ Note: Initialization by a trivial copy/move constructor is non-vacuous initialization. — end note ] 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) — if the object has non-vacuous initialization, its initialization is complete,

32 Under the “as-if” rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (6.8.1).
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 (11.6.1, 15.6.2), or as described in 12.3. The lifetime of an object \( o \) of type \( T \) ends when:

1. If \( T \) is a class type with a non-trivial destructor (15.4), the destructor call starts, or
2. The storage which the object occupies is released, or is reused by an object that is not nested within \( o \) (6.6.2).

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

2 [ Note: 15.6.2 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 15.6.2 and in 15.7. 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. 15.6.2 and 15.7 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 (8.5.2.5) 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 15.7. Otherwise, such a pointer refers to allocated storage (6.6.4.4.1), and using the pointer as if the pointer were of type \( \text{void*} \), is well-defined. Indirection through such a pointer is permitted but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

- 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,
- the pointer is used to access a non-static data member or call a non-static member function of the object, or
- the pointer is implicitly converted (7.11) to a pointer to a virtual base class, or
- the pointer is used as the operand of a static_cast (8.5.1.9), except when the conversion is to pointer to \( \text{cv void} \), or to pointer to \( \text{cv void} \) and subsequently to pointer to \( \text{cv char} \), \( \text{cv unsigned char} \), or \( \text{cv std::byte} \) (21.2.1), or
- the pointer is used as the operand of a dynamic_cast (8.5.1.7).

[ Example: ]

```cpp
#include <cstdlib>

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

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

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

33) For example, before the construction of a global object that is initialized via a user-provided constructor (15.7).
... = this;  // OK, this points to valid memory
}

void g() {
    void* p = std::malloc(sizeof(D1) + sizeof(D2));
    B* pb = new (p) D1;
    *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 15.7. Otherwise, such a glvalue refers to allocated storage (6.6.4.1), 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 (11.6.3), or
(7.4) — the glvalue is used as the operand of a \texttt{dynamic\_cast} (8.5.1.7) or as the operand of \texttt{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) — the original object was a most derived object (6.6.2) of type \texttt{T} and the new object is a most derived object of type \texttt{T} (that is, they are not base class subobjects).

\begin{example}
\begin{verbatim}
struct C {
    int i;
    void f();
    const C& operator=( const C& );
};

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

C c1;
C c2;
c1 = c2;  // well-defined
cl.f();  // well-defined; c1 refers to a new object of type C
\end{verbatim}
\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 \texttt{std::launder} (21.6). — end note]
If a program ends the lifetime of an object of type \( T \) with static (6.6.4.1), thread (6.6.4.2), or automatic (6.6.4.3) storage duration and if \( T \) has a non-trivial destructor, the program must ensure that an object of the original type occupies the same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [Example:

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

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

— end example]

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:

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

— end example]

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

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 (15.2). The dynamic storage duration is associated with objects created by a new-expression (8.5.2.4).

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.\(^\text{35}\)

\(\text{34}\) That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.

\(\text{35}\) Some implementations might define that copying an invalid pointer value causes a system-generated runtime fault.
6.6.4.1 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 15.8.

The keyword static can be used to declare a local variable with static storage duration. [Note: 9.7 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.

6.6.4.2 Thread storage duration

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.

6.6.4.3 Automatic storage duration

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

[Note: These variables are initialized and destroyed as described in 9.7. — 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 15.8.

6.6.4.4 Dynamic storage duration

Objects can be created dynamically during program execution (6.8.1), using new-expressions (8.5.2.4), and destroyed using delete-expressions (8.5.2.5). 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 21.6.2.3 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 (21.6.2). 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 (20.5.4.6). The following allocation and deallocation functions (21.6) are implicitly declared in global scope in each translation unit of a program.

[[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 (15.5).

Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the
library, shall conform to the semantics specified in 6.6.4.4.1 and 6.6.4.4.2.

### 6.6.4.4.1 Allocation functions

An allocation function shall be a class member function or a global function; a program is ill-formed if an
allocation function is declared in a namespace scope other than global scope or declared static in global
scope. The return type shall be void*. The first parameter shall have type std::size_t (21.2). The first
parameter shall not have an associated default argument (11.3.6). The value of the first parameter shall be
interpreted as the requested size of the allocation. An allocation function can be a function template. Such a
template shall declare its return type and first parameter as specified above (that is, template parameter
types shall not be used in the return type and first parameter type). Template allocation functions shall have
two or more parameters.

The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return
the address of the start of a block of storage whose length in bytes shall be at least as large as the requested
size. There are no constraints on the contents of the allocated storage on return from the allocation function.
The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are
unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer to any
suitable complete object type (21.6.2.1) 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 shall be a
non-null pointer value (7.11) p0 different from any previously returned value p1, unless that value p1 was
subsequently passed to an operator delete. Furthermore, for the library allocation functions in 21.6.2.1
and 21.6.2.2, p0 shall represent the address of a block of storage disjoint from the storage for any other object
accessible to the caller. The effect of indirecting through a pointer returned as a request for zero size is
undefined.\(^3\)

An allocation function that fails to allocate storage can invoke the currently installed new-handler function
(21.6.3.3), if any. [Note: A program-supplied allocation function can obtain the address of the currently
installed new_handler using the std::get_new_handler function (21.6.3.4). — end note] If an allocation
function that has a non-throwing exception specification (18.4) fails to allocate storage, it shall return a null
pointer. Any other allocation function that fails to allocate storage shall indicate failure only by throwing an
exception (18.1) of a type that would match a handler (18.3) of type std::bad_alloc (21.6.3.1).

A global allocation function is only called as the result of a new expression (8.5.2.4), or called directly using
the function call syntax (8.5.1.2), or called indirectly through calls to the functions in the C++ standard
library. [Note: In particular, a global allocation function is not called to allocate storage for objects with
static storage duration (6.6.4.1), for objects or references with thread storage duration (6.6.4.2), for objects
of type std::type_info (8.5.1.8), or for an exception object (18.1). — end note]

### 6.6.4.4.2 Deallocation functions

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

Each deallocation function shall return void and its first parameter shall be void*. A deallocation function
may have more than one parameter. A usual deallocation function is a deallocation function that has:

1. exactly one parameter; or
2. exactly two parameters, the type of the second being either std::align_val_t or std::size_t;\(^37\) or
3. exactly three parameters, the type of the second being std::size_t and the type of the third being

\(^3\) The intent is to have operator new() implementable by calling std::malloc() or std::calloc(), so the rules are substan-
tially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

\(^37\) The global operator delete(void*, std::size_t) precludes use of an allocation function void operator new(std::size_-
t, std::size_t) as a placement allocation function (C.3.2).
A deallocation function may be an instance of a function template. Neither the first parameter nor the return type shall depend on a template parameter. [Note: That is, a deallocation function template shall have a first parameter of type `void*` and a return type of `void` (as specified above). — end note] A deallocation function template shall have two or more function parameters. A template instance is never a usual deallocation function, regardless of its signature.

3 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.

4 If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (7.11), the deallocation function shall deallocate the storage referenced by the pointer, ending the duration of the region of storage.

### 6.6.4.4.3 Safely-derived pointers

A **traceable pointer object** is

1. an object of an object pointer type (6.7.2), or
2. an object of an integral type that is at least as large as `std::intptr_t`, or
3. a sequence of elements in an array of narrow character type (6.7.1), 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:

1. the value returned by a call to the C++ standard library implementation of `::operator new(std::size_t)` or `::operator new(std::size_t, std::align_val_t)`:\(^{38}\)
2. the result of taking the address of an object (or one of its subobjects) designated by an lvalue resulting from indirection through a safely-derived pointer value;
3. the result of well-defined pointer arithmetic (8.5.6) using a safely-derived pointer value;
4. the result of a well-defined pointer conversion (7.11, 8.5.3) of a safely-derived pointer value;
5. the result of a `reinterpret_cast` of a safely-derived pointer value;
6. the result of a `reinterpret_cast` of an integer representation of a safely-derived pointer value;
7. the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value.

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

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

An implementation may have **relaxed pointer safety**, in which case the validity of a pointer value does not depend on whether it is a safely-derived pointer value. Alternatively, an implementation may have **strict pointer safety**, in which case a pointer value referring to an object with dynamic storage duration that is not a safely-derived pointer value is an invalid pointer value unless the referenced complete object has previously been declared reachable (23.10.5). [Note: The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined, see 6.6.4. 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.

---

\(^{38}\) This subclause does not impose restrictions on indirection through pointers to memory not allocated by `::operator new`. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because indirection through pointers to memory allocated by `std::malloc` is not restricted.
6.6.4.5 Duration of subobjects

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

6.6.5 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 (10.6.2).

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)` (21.2). 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:

```c++
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)`. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (10.6.2). 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__` (19.8).

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 (8.5.2.6). Furthermore, the narrow character types (6.7.1) shall have the weakest alignment requirement. [Note: This enables the narrow character types to be used as the underlying type for an aligned memory area (10.6.2). — end note]

Comparing alignments is meaningful and provides the obvious results:

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

[Note: The runtime pointer alignment function (23.10.6) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (23.15.7.6) can be used to obtain aligned storage. — end note]

If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed.

6.7 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 (11.3.2), or functions (11.3.5). — end note]

For any object (other than a base-class 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 (21.2.1).\(^39\) If the content of that array is copied back into the object, the object shall subsequently hold its original value. [Example:

```c
#define N sizeof(T)
char buf[N]; // obj initialized to its original value
T obj; // between these two calls to std::memcpy, obj might be modified
std::memcpy(buf, &obj, N); // at this point, each subobject of obj of scalar type holds its original value
```

— 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 base-class subobject, if the underlying bytes (6.6.1) making up \(obj1\) are copied into \(obj2\), \(^40\) \(obj2\) shall subsequently hold the same value as \(obj1\). [Example:

```c
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 \(\text{sizeof}(T)\). The value representation of an object is the set of bits that hold the 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.\(^41\)

5 A class that has been declared but not defined, an enumeration type in certain contexts (10.2), or an array of unknown bound or of incomplete element type, is an incompletely-defined object type.\(^42\) 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;
```

```c
void foo() {
    xp++; // ill-formed: \(X\) is incomplete
    arrp++; // ill-formed: incomplete type
    arrpp++; // OK: sizeof UNKA* is known
}
```

```c
struct X { int i; }; // now X is a complete type
int arr[10]; // now the type of arr is complete
```

```c
X x;
void bar() {
    xp = &x; // OK; type is “pointer to \(X\)"
}
```

\(^39\) By using, for example, the library functions (20.5.1.2) `std::memcpy` or `std::memmove`.

\(^40\) By using, for example, the library functions (20.5.1.2) `std::memcpy` or `std::memmove`.

\(^41\) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.

\(^42\) The size and layout of an instance of an incompletely-defined object type is unknown.
An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not cv void.

A type is a literal type if it is:

- possibly cv-qualified void; or
- a scalar type; or
- a reference type; or
- an array of literal type; or
- a possibly cv-qualified class type (Clause 12) that has all of the following properties:
  - it has a trivial destructor,
  - it is either a closure type (8.4.5.1), an aggregate type (11.6.1), or has at least one constexpr constructor or constructor template (possibly inherited from a base class) that is not a copy or move constructor,
  - if it is a union, at least one of its non-static data members is of non-volatile literal type, and
  - if it is not a union, all of its non-static data members and base classes are of non-volatile literal types.

Note: A literal type is one for which it might be possible to create an object within a constant expression. It is not a guarantee that it is possible to create such an object, nor is it a guarantee that any object of that type will be usable in a constant expression. —end note]

Two types cv1 T1 and cv2 T2 are layout-compatible types if T1 and T2 are the same type, layout-compatible enumerations (10.2), or layout-compatible standard-layout class types (12.2).

6.7.1 Fundamental types [basic.fundamental]

Objects declared as characters (char) shall be large enough to store any member of the implementation’s basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a char object can hold negative values. Characters can be explicitly declared unsigned or signed. Plain char, signed char, and unsigned char are three distinct types, collectively called narrow character types. A char, a signed char, and an unsigned char occupy the same amount of storage and have the same alignment requirements (6.6.5); that is, they have the same object representation. For narrow character types, all bits of the object representation participate in the value representation. [Note: A bit-field of narrow character type whose length is larger than the number of bits in the object representation of that type has padding bits; see 6.7. — end note] For unsigned narrow character types, each possible bit pattern of the value representation represents a distinct number. These requirements do not hold for other types. In any particular implementation, a plain char object can take on either the same values as a signed char or an unsigned char; which one is implementation-defined. For each value i of type unsigned char in the range 0 to 255 inclusive, there exists a value j of type char such that the result of an integral conversion (7.8) from i to char is j, and the result of an integral conversion from j to unsigned char is i.

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. Plain ints have the natural size suggested by the architecture of the execution environment\(^43\); the other signed integer types are provided to meet special needs.

3 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”, each of which occupies the same amount of storage and has the same alignment requirements \((6.6.5)\) as the corresponding signed integer type\(^44\); that is, each signed integer type has the same object representation as its corresponding unsigned integer type. Likewise, for each of the extended signed integer types there exists a corresponding extended unsigned integer type with the same amount of storage and alignment requirements. The standard and extended unsigned integer types are collectively called unsigned integer types. The range of non-negative values of a signed integer type is a subrange of the corresponding unsigned integer type, the representation of the same value in each of the two types is the same, and the value representation of each corresponding signed/unsigned type shall be the same. 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. The signed and unsigned integer types shall satisfy the constraints given in the C standard, subclause 5.2.4.2.1.

4 Unsigned integers shall obey the laws of arithmetic modulo \(2^n\) where \(n\) is the number of bits in the value representation of that particular size of integer.\(^45\)

5 Type wchar_t is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales \((25.3.1)\). Type wchar_t shall have the same size, signedness, and alignment requirements \((6.6.5)\) as one of the other integral types, called its underlying type. Types char16_t and char32_t denote distinct types with the same size, signedness, and alignment as uint_least16_t and uint_least32_t, respectively, in \(<\text{cstdlib}>, \text{called the underlying types.}

6 Values of type bool are either true or false.\(^46\) [Note: There are no signed, unsigned, short, or long bool types or values. —end note] Values of type bool participate in integral promotions \((7.6)\).

7 Types bool, char, char16_t, char32_t, wchar_t, and the signed and unsigned integer types are collectively called integral types.\(^47\) A synonym for integral type is integer type. The representations of integral types shall define values by use of a pure binary numeration system.\(^48\) [Example: This document permits two’s complement, ones’ complement and signed magnitude representations for integral types. —end example]

8 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 21.3. —end note] Integral and floating types are collectively called arithmetic types. Specializations of the standard library template std::numeric_limits \((21.3)\) shall specify the maximum and minimum values of each arithmetic type for an implementation.

9 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 \((8.5.3)\). An expression of type cv void shall be used only as an expression statement \((9.2)\), as an operand of a comma expression \((8.5.19)\), as a second or third operand of ?: \((8.5.16)\), as the operand of typeid, noexcept, or decltype, as the expression in a return statement \((9.6.3)\) for a function with the return type cv void, or as the operand of an explicit conversion to type cv void.

10 A value of type std::nullptr_t is a null pointer constant \((7.11)\). Such values participate in the pointer and the pointer-to-member conversions \((7.11, 7.12)\). sizeof(std::nullptr_t) shall be equal to sizeof(void*).

\(^43\) int must also be large enough to contain any value in the range \([\text{INT_MIN, INT_MAX}]\), as defined in the header \(<\text{limits}\>.

\(^44\) See 10.1.7.2 regarding the correspondence between types and the sequences of type-specifiers that designate them.

\(^45\) This implies that unsigned arithmetic does not overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.

\(^46\) Using a bool value in ways described by this document as “undefined”, such as by examining the value of an uninitialized automatic object, might cause it to behave as if it is neither true nor false.

\(^47\) Therefore, enumerations \((10.2)\) are not integral; however, enumerations can be promoted to integral types as specified in 7.6.

\(^48\) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral power of 2, except perhaps for the bit with the highest position. (Adapted from the American National Dictionary for Information Processing Systems.)
Compartment types can be constructed in the following ways:

- arrays of objects of a given type, 11.3.4;
- functions, which have parameters of given types and return void or references or objects of a given type, 11.3.5;
- pointers to cv void or objects or functions (including static members of classes) of a given type, 11.3.1;
- references to objects or functions of a given type, 11.3.2. There are two types of references:
  - value reference
  - rvalue reference
- classes containing a sequence of objects of various types (Clause 12), a set of types, enumerations and functions for manipulating these objects (12.2.1), and a set of restrictions on the access to these entities (Clause 14);
- unions, which are classes capable of containing objects of different types at different times, 12.3;
- enumerations, which comprise a set of named constant values. Each distinct enumeration constitutes a different enumerated type, 10.2;
- pointers to non-static class members, which identify members of a given type within objects of a given class, 11.3.3. 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 11.3.1, 11.3.4, 11.3.5, and 11.3.2. Constructing a type such that the number of bytes in its object representation exceeds the maximum value representable in the type std::size_t (21.2) 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 (6.6.5). Every value of pointer type is one of the following:

- a pointer to an object or function (the pointer is said to point to the object or function), or
- a pointer past the end of an object (8.5.6), or
- the null pointer value (7.11) for that type, or
- 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 (8.5.6) 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 6.6.4. — end note] For purposes of pointer arithmetic (8.5.6) and comparison (8.5.9, 8.5.10), 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 (6.6.5). [Note: Pointers to over-aligned types (6.6.5) have no special representation, but their range of valid values is restricted by the extended alignment requirement. — end note]

Two objects a and b are pointer-interconvertible if:

- they are the same object, or

49) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.
50) For an object that is not within its lifetime, this is the first byte in memory that it will occupy or used to occupy.
one is a union object and the other is a non-static data member of that object (12.3), or

— 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, the first base class subobject of that object (12.2), or

— there exists an object \( c \) such that \( a \) and \( c \) are pointer-interconvertible, and \( c \) and \( b \) are pointer-interconvertible.

If two objects are pointer-interconvertible, then they have the same address, and it is possible to obtain a pointer to one from a pointer to the other via a \texttt{reinterpret_cast} (8.5.1.10). [Note: An array object and its first element are not pointer-interconvertible, even though they have the same address. —end note]

5 A pointer to \texttt{cv}-qualified (6.7.3) or \texttt{cv}-unqualified \texttt{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 \texttt{cv void*} shall have the same representation and alignment requirements as \texttt{cv char*}.

### 6.7.3 CV-qualifiers

1 A type mentioned in 6.7.1 and 6.7.2 is a \textit{cv-unqualified type}. Each type which is a \texttt{cv}-unqualified complete or incomplete object type or is \texttt{void} (6.7) has three corresponding \texttt{cv}-qualified versions of its type: a \texttt{const-qualified} version, a \texttt{volatile-qualified} version, and a \texttt{const-volatile-qualified} version. The type of an object (6.6.2) includes the \texttt{cv-qualifiers} specified in the \texttt{decl-specifier-seq} (10.1), \texttt{declarator} (Clause 11), \texttt{type-id} (11.1), or \texttt{new-type-id} (8.5.2.4) when the object is created.

1.1 A \textit{const object} is an object of type \texttt{const T} or a non-mutable subobject of such an object.

1.2 A \textit{volatile object} is an object of type \texttt{volatile T}, a subobject of such an object, or a mutable subobject of a const volatile object.

1.3 A \textit{const volatile object} is an object of type \texttt{const volatile T}, a non-mutable subobject of such an object, a const subobject of a volatile object, or a non-mutable volatile subobject of a const object.

The \texttt{cv}-qualified or \texttt{cv}-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (6.6.5).

2 A compound type (6.7.2) is not \texttt{cv}-qualified by the \texttt{cv-qualifiers} (if any) of the types from which it is compounded. Any \texttt{cv-qualifiers} applied to an array type affect the array element type (11.3.4).

3 See 11.3.5 and 12.2.2.1 regarding function types that have \texttt{cv-qualifiers}.

4 There is a partial ordering on \texttt{cv-qualifiers}, so that a type can be said to be \textit{more \texttt{cv}-qualified} than another. Table 10 shows the relations that constitute this ordering.

Table 10 — Relations on \textit{const} and \textit{volatile}

| no cv-qualifier \(<\) const          |
|---------------------------<----------------|
| no cv-qualifier \(<\) volatile   |
| no cv-qualifier \(<\) const volatile |
| const \(<\) const volatile      |
| volatile \(<\) const volatile    |

5 In this document, the notation \texttt{cv} (or \texttt{cv1}, \texttt{cv2}, etc.), used in the description of types, represents an arbitrary set of \texttt{cv}-qualifiers, i.e., one of \{\texttt{const}\}, \{\texttt{volatile}\}, \{\texttt{const, volatile}\}, or the empty set. For a type \texttt{cv T}, the \textit{top-level \texttt{cv}-qualifiers} of that type are those denoted by \texttt{cv}. [Example: The type corresponding to the \texttt{type-id \texttt{const int}*} has no top-level \texttt{cv}-qualifiers. The type corresponding to the \texttt{type-id \texttt{volatile int *\textit{const}}} has the top-level \texttt{cv}-qualifier \texttt{const}. For a class type \texttt{C}, the type corresponding to the \texttt{type-id \texttt{void (C::*: volatile)\textit{(int) const}}} has the top-level \texttt{cv}-qualifier \texttt{volatile}. —end example]

6 \texttt{cv}-qualified applied to an array type attach to the underlying element type, so the notation \texttt{“cv T”}, where \texttt{T} is an array type, refers to an array whose elements are so-qualified. An array type whose elements are \texttt{cv}-qualified is also considered to have the same \texttt{cv}-qualifications as its elements. [Example:

```c
typedef char CA[5];
typedef const char CC;
CC arr1[5] = { 0 };
const CA arr2 = { 0 };
```

51) 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.

§ 6.7.3 62
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 [conv.rank]

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

(1.1) — No two signed integer types other than char and signed char (if char is signed) shall have the same rank, even if they have the same representation.

(1.2) — The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.

(1.3) — The rank of long long int shall be greater than the rank of long int, which shall be greater than the rank of int, which shall be greater than the rank of short int, which shall be greater than the rank of signed char.

(1.4) — The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.

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

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

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

(1.8) — The ranks of char16_t, char32_t, and wchar_t shall equal the ranks of their underlying types (6.7.1).

(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.

(1.10) — For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 shall have greater rank than T3.

[Note: The integer conversion rank is used in the definition of the integral promotions (7.6) and the usual arithmetic conversions (8.2). — end note]

6.8 Program execution [basic.exec]

6.8.1 Sequential execution [intro.execution]

1 An instance of each object with automatic storage duration (6.6.4.3) 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 or receipt of a signal).

2 A constituent expression is defined as follows:

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

(2.2) — The constituent expressions of a braced-init-list or of a (possibly parenthesized) expression-list are the constituent expressions of the elements of the respective list.

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

[Example:

struct A { int x; };
struct B { int y; struct A a; };
B b = { 5, { 1+1 } };]

The constituent expressions of the initializer used for the initialization of b are 5 and 1+1. — end example]

3 The immediate subexpressions of an expression e are

(3.1) — the constituent expressions of e’s operands (8.2),

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

(3.3) — if e is a lambda-expression (8.4.5), the initialization of the entities captured by copy and the constituent expressions of the initializer of the init-captures,

(3.4) — if e is a function call (8.5.1.2) or implicitly invokes a function, the constituent expressions of each default argument (11.3.6) used in the call, or
if \( e \) creates an aggregate object (11.6.1), the constituent expressions of each default member initializer (12.2) used in the initialization.

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

A *full-expression* is

1. an unevaluated operand (8.2),
2. a *constant-expression* (8.6),
3. an *init-declarator* (Clause 11) or a *mem-initializer* (15.6.2), including the constituent expressions of the initializer,
4. an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object (15.2), or
5. 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:

```cpp
struct S {
    S(int i): I(i) { }           // full-expression is initialization of I
    int& v() { return I; }     //S() noexcept(false) { }
private:
    int I;
};
S s1(1);                    // full-expression comprises call of S::S(int)
void f() {                  // full-expression comprises call of S::S(int)
    S s2 = 2;               // full-expression includes lvalue-to-rvalue and int to bool conversions,
    if (S(3).v())           // 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));           // full-expression is the entire initialization
};
B b[2] = { B(), B() };     // including the destruction of temporaries
```

— end example]

[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 (11.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — end note]

Reading an object designated by a *volatile* glvalue (8.2.1), 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.52

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++ + i;             // the behavior is undefined
    i = i + 1;               // the value of i is incremented
}
—end example]
```

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.53 [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 8.5.2.4. For another example, invocation of a conversion function (15.3.2) can arise in contexts in which no function call syntax appears. —end example] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

12 If a signal handler is executed as a result of a call to the `std::raise` function, then the execution of the handler is sequenced after the invocation of the `std::raise` function and before its return. [Note: When a signal is received for another reason, the execution of the signal handler is usually unsequenced with respect to the rest of the program. —end note]

### 6.8.2 Multi-threaded executions and data races

A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [Note: When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. —end note] Every thread in a

52) As specified in 15.2, 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.

53) In other words, function executions do not interleave with each other.
program can potentially access every object and function in a program. Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads. [Note: Usually the execution can be viewed as an interleaving of all of its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — end note] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

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

### 6.8.2.1 Data races

1. The value of an object visible to a thread $T$ at a particular point is the initial value of the object, a value assigned to the object by $T$, or a value assigned to the object by another thread, according to the rules below. [Note: 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 32) and operations on mutexes (Clause 33) 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 subsequence of side effects in the modification order of $M$, where the first operation is $A$, and every subsequent operation

   (5.1) — is performed by the same thread that performed $A$, or

   (5.2) — 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 (32.4). [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]

7. An evaluation $A$ carries a dependency to an evaluation $B$ if

   (7.1) — the value of $A$ is used as an operand of $B$, unless:

   (7.1.1) — $B$ is an invocation of any specialization of std::kill_dependency (32.4), or

   (7.1.2) — $A$ is the left operand of a built-in logical AND ($\&\&$, see 8.5.14) or logical OR ($||$, see 8.5.15) operator, or

54 An object with automatic or thread storage duration (6.6.4) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (6.7.2).
(7.1.3) — A is the left operand of a conditional (?) operator, or
(7.1.4) — A is the left operand of the built-in comma (,) operator (8.5.19);
or
(7.2) — A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or
(7.3) — for some evaluation X, A carries a dependency to X, and X carries a dependency to B.
[Note: “Carries a dependency to” is a subset of “is sequenced before”, and is similarly strictly intra-thread. —end note]

8 An evaluation A is dependency-ordered before an evaluation B if
(8.1) — A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or
(8.2) — 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]

9 An evaluation A inter-thread happens before an evaluation B if
(9.1) — A synchronizes with B, or
(9.2) — A is dependency-ordered before B, or
(9.3) — for some evaluation X
(9.3.1) — A synchronizes with X and X is sequenced before B, or
(9.3.2) — A is sequenced before X and X inter-thread happens before B, or
(9.3.3) — A inter-thread happens before X and X inter-thread happens before B.
[Note: The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced before”, “synchronizes with” and “dependency-ordered before” relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with “dependency-ordered before” followed by “sequenced before”. The reason for this limitation is that a consume operation participating in a “dependency-ordered before” relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of “sequenced before”. The reasons for this limitation are (1) to permit “inter-thread happens before” to be transitively closed and (2) the “happens before” relation, defined below, provides for relationships consisting entirely of “sequenced before”. —end note]

10 An evaluation A happens before an evaluation B (or, equivalently, B happens after A) if:
(10.1) — A is sequenced before B, or
(10.2) — A inter-thread happens before B.
The implementation shall ensure that no program execution demonstrates a cycle in the “happens before” relation. [Note: This cycle would otherwise be possible only through the use of consume operations. —end note]

11 An evaluation A strongly happens before an evaluation B if either
(11.1) — A is sequenced before B, or
(11.2) — A synchronizes with B, or
(11.3) — A strongly happens before X and X strongly happens before B.
[Note: In the absence of consume operations, the happens before and strongly happens before relations are identical. Strongly happens before essentially excludes consume operations. —end note]

12 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:
(12.1) — A happens before B and
(12.2) — 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]

13 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]

14 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]

15 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]

16 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]

17 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]

18 [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]

19 [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]

20 Two actions are potentially concurrent if

(20.1) — they are performed by different threads, or

(20.2) — 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 that programs that correctly use atomics 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]

21 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 \( B \). [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]

23 [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 [intro.progress]

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

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

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

2 Executions of atomic functions that are either defined to be lock-free (32.8) or indicated as lock-free (32.5) are lock-free executions.

(2.1) — If there is only one thread that is not blocked (3.6) in a standard library function, a lock-free execution in that thread shall complete. [Note: Concurrently executing threads may prevent progress of a lock-free execution. For example, this situation can occur with load-locked store-conditional implementations. This property is sometimes termed obstruction-free. —end note]

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

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

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

4 An invocation of a standard library function that blocks (3.6) is considered to continuously execute execution steps while waiting for the condition that it blocks on to be satisfied. [Example: A library I/O function that blocks until the I/O operation is complete can be considered to continuously check whether the operation is complete. Each such check might consist of one or more execution steps, for example using observable behavior of the abstract machine. —end example]

5 [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]

6 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).

7 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(33.3.2)` 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 to strengthen a particular thread of execution whose too-weak progress guarantee is preventing overall progress. —end note]

An implementation should ensure that the last value (in modification order) assigned by an atomic or synchronization operation will become visible to all other threads in a finite period of time.

### 6.8.3 Start and termination

#### 6.8.3.1 main function

A program shall contain a global function called `main`. Executing a program starts a main thread of execution (6.8.2, 33.3) 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` (11.3.5). 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) (20.4.2.1.5.2) 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 `main` function shall not be declared with a `linkage-specification` (10.5). A program that declares a variable `main` at global scope 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)` (21.5)) does not destroy any objects with automatic storage duration (15.4). 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 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 18.3).

### 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.

A constant initializer for a variable or temporary object `o` is an initializer whose full-expression is 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. — end note] Constant initialization is performed if a variable or temporary object with static or thread storage duration is initialized by a constant initializer for the entity. If constant initialization is not performed, a variable with static storage duration (6.6.4.1) or thread storage duration (6.6.4.2) is zero-initialized (11.6). 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 9.7. — 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,

```
inline double fd() { return 1.0; }
extern double d1;
```
Dynamic initialization of non-local variables

1 Dynamic initialization of a non-local variable with static storage duration is unordered if the variable is an implicitly or explicitly instantiated specialization, is partially-ordered if the variable is an inline variable that is not an implicitly or explicitly instantiated specialization, and otherwise is ordered. [Note: An explicitly specialized non-inline static data member or variable template specialization has ordered initialization. — end note]

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

\[\begin{align*}
(2.1) & \quad \text{If } V \text{ and } W \text{ have ordered initialization and } V \text{ is defined before } W \text{ within a single translation unit, the} \\
& \quad \text{initialization of } V \text{ is sequenced before the initialization of } W. \\
(2.2) & \quad \text{If } V \text{ has partially-ordered initialization, } W \text{ does not have unordered initialization, and } V \text{ is defined before} \\
& \quad W \text{ in every translation unit in which } W \text{ is defined, then} \\
(2.2.1) & \quad \text{if the program starts a thread (6.8.2) other than the main thread (6.8.3.1), the initialization of } V \text{ strongly happens before the initialization of } W; \\
(2.2.2) & \quad \text{otherwise, the initialization of } V \text{ is sequenced before the initialization of } W. \\
(2.3) & \quad \text{Otherwise, if the program starts a thread other than the main thread before either } V \text{ or } W \text{ is initialized,} \\
& \quad \text{it is unspecified in which threads the initializations of } V \text{ and } W \text{ occur; the initializations are unsequenced} \\
& \quad \text{if they occur in the same thread.} \\
(2.4) & \quad \text{Otherwise, the initializations of } V \text{ and } W \text{ are indeterminately sequenced.} \\
\end{align*}\]

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

3 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.

4 It is implementation-defined whether the dynamic initialization of a non-local non-inline variable with static storage duration is sequenced before the first statement of \texttt{main} or is deferred. If it is deferred, it strongly happens before any non-initialization odr-use of any non-inline function or non-inline variable defined in the same translation unit as the variable to be initialized.\footnote{\texttt{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, 6.6.4.1).}} 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:

// - File 1 -
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
}

// - File 2 -
#include "a.h"
A a;

// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
extern B b;
int main() {
  a.Use();
  b.Use();
}

It is implementation-defined whether either a or b is initialized before main is entered or whether the
initializations are delayed until a is first odr-used in main. In particular, if a is initialized before main is
entered, it is not guaranteed that b will be initialized before it is odr-used by the initialization of a, that is,
before A::A is called. If, however, a is initialized at some point after the first statement of main, b will be
initialized prior to its use in A::A. — end example

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 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
defered, the initialization associated with the entity for thread t is sequenced before the first non-initialization
odr-use by t of any non-inline variable with thread storage duration defined in the same translation unit
as the variable to be initialized. It is implementation-defined in which threads and at which points in the
program such deferred dynamic initialization occurs.

7 If the initialization of a non-local variable with static or thread storage duration exits via an exception,
std::terminate is called (18.5.1).

6.8.3.4 Termination [basic.start.term]

1 Destructors (15.4) for initialized objects (that is, objects whose lifetime (6.6.3) has begun) with static storage
duration, and functions registered with std::atexit, are called as part of a call to std::exit (21.5). The
call to std::exit is sequenced before the invocations of the destructors and the registered functions. [Note:
Returning from main invokes std::exit (6.8.3.1). — end note]

2 Destructors for initialized objects with thread storage duration within a given thread are called as a result
of returning from the initial function of that thread and as a result of that thread calling std::exit. The
completions of the destructors for all initialized objects with thread storage duration within that thread
strongly happen before the initiation of the destructors of 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, std::terminate is called (18.5.1).

4 If a function contains a block-scope object of static or thread storage duration that has been destroyed and the
function is called during the destruction of an object with static or thread storage duration, the program has
undefined behavior if the flow of control passes through the definition of the previously destroyed block-scope
object. Likewise, the behavior is undefined if the block-scope object is used indirectly (i.e., through a pointer)
after its destruction.

5 If the completion of the initialization of an object with static storage duration strongly happens before a call
to std::atexit (see <cstdlib>, 21.5), 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.

6 If there is a use of a standard library object or function not permitted within signal handlers (21.11) that
does not happen before (6.8.2) completion of destruction of objects with static storage duration and execution
of std::atexit registered functions (21.5), 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

§ 6.8.3.4
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]

7 Calling the function `std::abort()` declared in `<stdlib>` terminates the program without executing any destructors and without calling the functions passed to `std::atexit()` or `std::at_quick_exit()`.
Standard conversions

1 Standard conversions are implicit conversions with built-in meaning. Clause 7 enumerates the full set of such conversions. A *standard conversion sequence* is a sequence of standard conversions in the following order:

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

1.2 Zero or one conversion from the following set: integral promotions, floating-point promotion, integral conversions, floating-point conversions, floating-integral conversions, pointer conversions, pointer-to-member conversions, and boolean conversions.

1.3 Zero or one function pointer conversion.

1.4 Zero or one qualification conversion.

[Note: A standard conversion sequence can be empty, i.e., it can consist of no conversions. —end note]

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

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

2.1 When used as operands of operators. The operator’s requirements for its operands dictate the destination type (8.5).

2.2 When used in the condition of an if statement or iteration statement (9.4, 9.5). The destination type is bool.

2.3 When used in the expression of a switch statement. The destination type is integral (9.4).

2.4 When used as the source expression for an initialization (which includes use as an argument in a function call and use as the expression in a return statement). The type of the entity being initialized is (generally) the destination type. See 11.6, 11.6.3.

—end note]

3 An expression e can be *implicitly converted* to a type T if and only if the declaration T t=e; is well-formed, for some invented temporary variable t (11.6).

4 Certain language constructs require that an expression be converted to a Boolean value. An expression e appearing in such a context is said to be *contextually converted to bool* and is well-formed if and only if the declaration bool t(e); is well-formed, for some invented temporary variable t (11.6).

5 Certain language constructs require conversion to a value having one of a specified set of types appropriate to the construct. An expression e of class type 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.

6 The effect of any implicit conversion is the same as performing the corresponding declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if T is an lvalue reference type or an rvalue reference to function type (11.3.2), 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.

[Note: For class types, user-defined conversions are considered as well; see 15.3. In general, an implicit conversion sequence (16.3.3.1) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. —end note]

7 [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]
7.1 Lvalue-to-rvalue conversion \[\text{conv.lval}\]

1 A glvalue (8.2.1) of a non-function, non-array type \(T\) can be converted to a prvalue.\(^{56}\) 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 \(T\).\(^{57}\)

2 When an lvalue-to-rvalue conversion is applied to an expression \(e\), and either
   \begin{itemize}
   \item \((2.1)\) \(e\) is not potentially evaluated, or
   \item \((2.2)\) 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{itemize}
the value contained in the referenced object is not accessed. \[\text{Example:} \]

   \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}

   \[\text{— end example} \]

3 The result of the conversion is determined according to the following rules:
   \begin{itemize}
   \item \((3.1)\) If \(T\) is \(\text{cv std::nullptr_t}\), the result is a null pointer constant (7.11). \[\text{Note:} \] Since no value is fetched from memory, there is no side effect for a volatile access (6.8.1), and an inactive member of a union (12.3) may be accessed. \text{— end note} \]
   \item \((3.2)\) Otherwise, if \(T\) has a class type, the conversion copy-initializes the result object from the glvalue.
   \item \((3.3)\) Otherwise, if the object to which the glvalue refers contains an invalid pointer value (6.6.4.4.2, 6.6.4.4.3), the behavior is implementation-defined.
   \item \((3.4)\) Otherwise, the value contained in the object indicated by the glvalue is the prvalue result.
   \end{itemize}

[\text{Note:} See also 8.2.1. \text{— end note}]

7.2 Array-to-pointer conversion \[\text{conv.array}\]

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 (7.4) is applied. The result is a pointer to the first element of the array.

7.3 Function-to-pointer conversion \[\text{conv.func}\]

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.\(^{58}\)

2 [\text{Note:} See 16.4 for additional rules for the case where the function is overloaded. \text{— end note}]

7.4 Temporary materialization conversion \[\text{conv.rval}\]

1 A prvalue of type \(T\) can be converted to an xvalue of type \(T\). This conversion initializes a temporary object (15.2) of \(T\) 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. [\text{Note:} If \(T\) is a class type (or array thereof), it must have an accessible and non-deleted destructor; see 15.4. \text{— end note}]

[\text{Example:}]

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

\[\text{— end example} \]

\(^{56}\) For historical reasons, this conversion is called the “lvalue-to-rvalue” conversion, even though that name does not accurately reflect the taxonomy of expressions described in 8.2.1.

\(^{57}\) In C++ class and array prvalues can have cv-qualified types. This differs from ISO C, in which non-lvalues never have cv-qualified types.

\(^{58}\) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.
7.5 Qualification conversions

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

\[
\text{"cv}_0 \ P_0 \ \text{"cv}_1 \ P_1 \ \ldots \ \text{"cv}_{n-1} \ P_{n-1} \ \text{"cv}_n \ U\]
\]

for \( n > 0 \), where each \( \text{"cv}_i \) is a set of cv-qualifiers (6.7.3), and each \( P_i \) is “pointer to” (11.3.1), “pointer to member of class \( C \) of type” (11.3.3), “array of \( N_i \)”, or “array of unknown bound of” (11.3.4). 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 expression of type \( T_1 \) can be converted to type \( T_2 \) if the following conditions are satisfied, where \( cv_i^j \) denotes the cv-qualifiers in the cv-qualification signature of \( T_j \).

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

[Note: If a program could assign a pointer of type \( T** \) to a pointer of type const \( T** \) (that is, if line #1 below were allowed), a program could inadvertently modify a const object (as it is done on line #2). For example,

```c
int main() {
    const char c = 'c';
    char* pc;
    const char** pcc = &pc;    // #1: not allowed
    *pcc = &c;
    *pc = 'C';    // #2: modifies a const object
}

— end note]

A prvalue of type “pointer to cv1 T” can be converted to a prvalue of type “pointer to cv2 T” if “cv2 T” is more cv-qualified than “cv1 T”. A prvalue of type “pointer to member of \( X \) of type cv1 T” can be converted to a prvalue of type “pointer to member of \( X \) of type cv2 T” if “cv2 T” is more cv-qualified than “cv1 T”. —end note]

[Note: Function types (including those used in pointer to member function types) are never cv-qualified (11.3.5). —end note]

7.6 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 (10.2) can be converted to a prvalue of the first of the following types that can represent all the values of the enumeration (i.e., the values in the range \( b_{\text{min}} \) to \( b_{\text{max}} \) as described in 10.2): 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 int 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 (10.2) can be converted to a prvalue of its underlying type. Moreover, if integral promotion can be applied to its underlying type, a

---

59) These rules ensure that const-safety is preserved by the conversion.
prvalue of an unscoped enumeration type whose underlying type is fixed can also be converted to a prvalue of the promoted underlying type.

5 A prvalue for an integral bit-field (12.2.4) can be converted to a prvalue of type int if int can represent all the values of the bit-field; otherwise, it can be converted to unsigned int if unsigned int can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

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

7 These conversions are called integral promotions.

7.7 Floating-point promotion [conv.fpprom]

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

2 This conversion is called floating-point promotion.

7.8 Integral conversions [conv.integral]

1 A prvalue of an integer type can be converted to a prvalue of another integer type. A prvalue of an unscoped enumeration type can be converted to a prvalue of an integer type.

2 If the destination type is unsigned, the resulting value is the least unsigned integer congruent to the source integer (modulo $2^n$ where $n$ is the number of bits used to represent the unsigned type). [Note: In a two’s complement representation, this conversion is conceptual and there is no change in the bit pattern (if there is no truncation). —end note]

3 If the destination type is signed, the value is unchanged if it can be represented in the destination type; otherwise, the value is implementation-defined.

4 If the destination type is bool, see 7.14. If the source type is bool, the value false is converted to zero and the value true is converted to one.

5 The conversions allowed as integral promotions are excluded from the set of integral conversions.

7.9 Floating-point conversions [conv.double]

1 A prvalue of floating-point type can be converted to a prvalue of another floating-point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.

2 The conversions allowed as floating-point promotions are excluded from the set of floating-point conversions.

7.10 Floating-integral conversions [conv.fpint]

1 A prvalue of a floating-point type can be converted to a prvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [Note: If the destination type is bool, see 7.14. —end note]

2 A prvalue of an integer type or of an unscoped enumeration type can be converted to a prvalue of a floating-point type. The result is exact if possible. If the value being converted is in the range of values that can be represented but the value cannot be represented exactly, it is an implementation-defined choice of either the next lower or higher representable value. [Note: Loss of precision occurs if the integral value cannot be represented exactly as a value of the floating type. —end note] If the value being converted is outside the range of values that can be represented, the behavior is undefined. If the source type is bool, the value false is converted to zero and the value true is converted to one.

7.11 Pointer conversions [conv.ptr]

1 A null pointer constant is an integer literal (5.13.2) with value zero or a prvalue of type std::nullptr_t. A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of object pointer or function pointer type. Such a conversion is called a null pointer conversion. Two null pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (7.5). A null pointer constant of integral type can
be converted to a prvalue of type `std::nullptr_t`. [Note: The resulting prvalue is not a null pointer value. — end note]

2 A prvalue of type “pointer to cv T”, where T is an object type, can be converted to a prvalue of type “pointer to cv void”. The pointer value (6.7.2) is unchanged by this conversion.

3 A prvalue of type “pointer to cv D”, where D is a class type, can be converted to a prvalue of type “pointer to cv B”, where B is a base class (Clause 13) of D. If B is an inaccessible (Clause 14) or ambiguous (13.2) 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.

7.12 Pointer-to-member conversions

A null pointer constant (7.11) 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 (7.5).

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 derived class (Clause 13) of B. If B is an inaccessible (Clause 14), ambiguous (13.2), or virtual (13.1) 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 is a pointer to member of cv-qualified type; indirecting through it with a D object is valid. The result is the same as if indirecting through the pointer to member of B with the B subobject of D. The null member pointer value is converted to the null member pointer value of the destination type.

7.13 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 points to the member function.

[Example:

```c++
void (*p)();
void (**pp)() noexcept = &p;  // error: cannot convert to pointer to noexcept function

struct S { typedef void (*p)(); operator p(); };
void (*q)() noexcept = S();   // error: cannot convert to pointer to noexcept function
```
— end example]

7.14 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. For direct-initialization (11.6), a prvalue of type `std::nullptr_t` can be converted to a prvalue of type bool; the resulting value is false.

60) 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) (7.11, Clause 13). 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*.
8 Expressions

8.1 Preamble

1 [Note: Clause 8 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]

2 [Note: Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 12) or enumeration type (10.2). Uses of overloaded operators are transformed into function calls as described in 16.5. Overloaded operators obey the rules for syntax and evaluation order specified in 8.5, 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 (16.5). — end note]

3 Subclause 8.5 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 8.5; see 16.3.1.2, 16.6.

4 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]

5 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.

8.2 Properties of expressions

8.2.1 Value category

1 Expressions are categorized according to the taxonomy in Figure 1.

expression
  ┌────────────────┐
  │ glvalue ──────┘rvalue
  │               │
  │ lvalue ───────┘xvalue ───────prvalue

Figure 1 — Expression category taxonomy

(1.1) — A glvalue is an expression whose evaluation determines the identity of an object, bit-field, or function.

(1.2) — A prvalue is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator, as specified by the context in which it appears.

(1.3) — An xvalue is a glvalue that denotes an object or bit-field whose resources can be reused (usually because it is near the end of its lifetime). [Example: Certain kinds of expressions involving rvalue references (11.3.2) yield xvalues, such as a call to a function whose return type is an rvalue reference or a cast to an rvalue reference type. — end example]

(1.4) — An lvalue is a glvalue that is not an xvalue.

(1.5) — An rvalue is a prvalue or an xvalue.

61) The precedence of operators is not directly specified, but it can be derived from the syntax.

62) The cast and assignment operators must still perform their specific conversions as described in 8.5.3, 8.5.1.9 and 8.5.18.
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 8.5 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]

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]

An expression is an xvalue if it is:

1. the result of calling a function, whether implicitly or explicitly, whose return type is an rvalue reference to object type,
2. a cast to an rvalue reference to object type,
3. a class member access expression designating a non-static data member of non-reference type in which the object expression is an xvalue, or
4. a .* pointer-to-member expression in which the first operand is an xvalue and the second operand is a pointer to data member.

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]

The result of a prvalue is the value that the expression stores into its context. 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 prvalue that is used to compute the value of an operand of an operator or 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, a temporary object is materialized; see 8.2. —end note]

Unless otherwise indicated (8.5.1.2), a prvalue shall always have complete type or the void type. A glvalue shall not have type cv void. [Note: A glvalue may have complete or incomplete non-void type. Class and array prvalues can have cv-qualified types; other prvalues always have cv-unqualified types. See 8.2. —end note]
An lvalue is *modifiable* unless its type is const-qualified or is a function type. [Note: A program that attempts to modify an object through a nonmodifiable lvalue expression or through an rvalue expression is ill-formed (8.5.18, 8.5.1.6, 8.5.2.2). — end note]

If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:63

- the dynamic type of the object,
- a cv-qualified version of the dynamic type of the object,
- a type similar (as defined in 7.5) to the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
- an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),
- a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
- a cv-qualified version of the dynamic type of the object,
- an aggregate or union type that includes one of the aforementioned types among its elements or non-static data members (including, recursively, an element or non-static data member of a subaggregate or contained union),
- a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
- a char, unsigned char, or std::byte type.

8.2.2 Type [expr.type]

If an expression initially has the type "reference to T" (11.3.2, 11.6.3), 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 rvalue, 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 "reference to function", 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 (7.5) is:

- for every i > 0, cv\_i\_T is the union of cv\_i\_T1 and cv\_i\_T2;
- if the resulting cv\_i\_T is different from cv\_i\_T1 or cv\_i\_T2, then const is added to every cv\_k\_T for 0 < k < i.

[Note: Given similar types T1 and T2, this construction ensures that both can be converted to T3. — end note]

The composite pointer type of two operands p1 and p2 having types T1 and T2, respectively, where at least one is a pointer or pointer-to-member type or std::nullptr\_t, is:

- if both p1 and p2 are null pointer constants, std::nullptr\_t;
- if either p1 or p2 is a null pointer constant, T2 or T1, respectively;
- if T1 or T2 is "pointer to cv1 void" and the other type is "pointer to cv2 T", where T is an object type or void, “pointer to cv12 void", where cv12 is the union of cv1 and cv2;
- if T1 or T2 is “pointer to noexcept function" and the other type is “pointer to function", where the function types are otherwise the same, “pointer to function";
- if T1 is “pointer to cv1 C1" and T2 is “pointer to cv2 C2", where C1 is reference-related to C2 or C2 is reference-related to C1 (11.6.3), the cv-combined type of T1 and T2 or the cv-combined type of T2 and T1, respectively;
- if T1 is “pointer to member of C1 of type cv1 U1" and T2 is “pointer to member of C2 of type cv2 U2" where C1 is reference-related to C2 or C2 is reference-related to C1 (11.6.3), the cv-combined type of T2 and T1 or the cv-combined type of T1 and T2, respectively;
- otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.

[Example:

typedef void *p;

63] The intent of this list is to specify those circumstances in which an object may or may not be aliased.
typedef const int *q;
typedef int **pi;
typedef const int **pci;

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

8.2.3 Context dependence

1 In some contexts, unevaluated operands appear \( (8.4.7, 8.5.1.8, 8.5.2.3, 8.5.2.7, 10.1.7.2, \text{Clause 17}) \). An unevaluated operand is not evaluated. \[\text{Note: In an unevaluated operand, a non-static class member may be named (8.4) and naming of objects or functions does not, by itself, require that a definition be provided (6.2). An unevaluated operand is considered a full-expression (6.8.1).} \text{-end note}\]

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 (7.2) and function-to-pointer (7.3) standard conversions are not applied. The lvalue-to-rvalue conversion (7.1) 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 (8.4.4)},
\item subscripting (8.5.1.1),
\item class member access (8.5.1.5),
\item indirection (8.5.2.1),
\item pointer-to-member operation (8.5.4),
\item conditional expression (8.5.16) where both the second and the third operands are one of these expressions, or
\item comma expression (8.5.19) where the right operand is one of these expressions.
\end{enumerate}

\[\text{Note: Using an overloaded operator causes a function call; the above covers only operators with built-in meaning.} \text{-end note}\] If the (possibly converted) expression is a prvalue, the temporary materialization conversion (7.4) is applied. \[\text{Note: If the expression is an lvalue of class type, it must have a volatile copy constructor to initialize the temporary object that is the result object of the lvalue-to-rvalue conversion.} \text{-end note}\] The glvalue expression is evaluated and its value is discarded.

8.3 Usual arithmetic conversions

1 Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows:

\begin{enumerate}
\item If either operand is of scoped enumeration type (10.2), no conversions are performed; if the other operand does not have the same type, the expression is ill-formed.
\item If either operand is of type \texttt{long double}, the other shall be converted to \texttt{long double}.
\item Otherwise, if either operand is \texttt{double}, the other shall be converted to \texttt{double}.
\item Otherwise, if either operand is \texttt{float}, the other shall be converted to \texttt{float}.
\item Otherwise, the integral promotions (7.6) shall be performed on both operands.\textsuperscript{64} Then the following rules shall be applied to the promoted operands:
\item If both operands have the same type, no further conversion is needed.
\item 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.
\item 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.
\end{enumerate}

\textsuperscript{64} As a consequence, operands of type \texttt{bool, char16_t, char32_t, wchar_t}, or an enumerated type are converted to some integral type.

\section*{§ 8.3}
— Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.

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

8.4 Primary expressions

primary-expression:
literal
this
(expression)
?id-expression
lambda-expression
fold-expression
requires-expression

8.4.1 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.

8.4.2 This

The keyword this names a pointer to the object for which a non-static member function (12.2.2.1) is invoked or a non-static data member’s initializer (12.2) 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).

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

```c
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]  

Otherwise, if a member-declarator declares a non-static data member (12.2) of a class X, the expression this is a prvalue of type “pointer to X” within the optional default member initializer (12.2). It shall not appear elsewhere in the member-declarator.

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

```c
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]
8.4.3 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.

8.4.4 Names

id-expression:
  unqualified-id
  qualified-id

An id-expression is a restricted form of a primary-expression. [Note: An id-expression can appear after . and -> operators (8.5.1.5). — 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 (8.5.1.5) 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 (8.5.2.1), or
3. if that id-expression denotes a non-static data member and it appears in an unevaluated operand. [Example:

```c
struct S {
  int m;
};
int i = sizeof(S::m); // OK
int j = sizeof(S::m + 42); // OK
```

— end example]

An id-expression that denotes the specialization of a concept (17.6.8) results in a prvalue of type bool. The expression is true if the concept’s normalized constraint-expression (17.4.2) is satisfied (17.4.1) by the specified template arguments and false otherwise. [Example:

```c
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 (17.2) and during overload resolution (Clause 16), and they are compared during the the partial ordering of constraints (17.4.4). — 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:

```c
void f(int) requires false;
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 (8.2). — end example]

8.4.4.1 Unqualified names

unqualified-id:
  identifier
  operator-function-id
  conversion-function-id
  literal-operator-id
~ class-name
~ decltype-specifier
template-id

---

65) This also applies when the object expression is an implicit (*this) (12.2.2).
An identifier is an id-expression provided it has been suitably declared (Clause 10). [Note: For operator-function-ids, see 16.5; for conversion-function-ids, see 15.3.2; for literal-operator-ids, see 16.5.8; for template-ids, see 17.2. A class-name or decltype-specifier prefixed by ~ denotes a destructor; see 15.4. Within the definition of a non-static member function, an identifier that names a non-static member is transformed to a class member access expression (12.2.2). — end note]

The result is the entity denoted by the identifier. If the entity is a local entity and naming it from outside of an unevaluated operand within the declarative region where the unqualified-id appears would result in some intervening lambda-expression capturing it by copy (8.4.5.2), the type of the expression is the type of a class member access expression (8.5.1.5) naming the non-static data member that would be declared for such a capture in the closure object of the innermost such intervening lambda-expression. [Note: If that lambda-expression is not declared mutable, the type of such an identifier will typically be const qualified. — end note] Otherwise, the type of the expression is the type of the result. [Note: The type will be adjusted as described in 8.2.2 if it is cv-qualified or is a reference type. — end note] The expression is an lvalue if the entity is a function, variable, or data member and a prvalue otherwise (8.2.1); it is a bit-field if the identifier designates a bit-field (11.5). [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]
```

§ 8.4.4.2  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.

2 A nested-name-specifier that denotes a class, optionally followed by the keyword template (17.2), and then followed by the name of a member of either that class (12.2) or one of its base classes (Clause 13), 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 class-name ::- class-name is used, the two class-names shall refer to the same class; this notation names the destructor (15.4). The form ~ decltype-specifier also denotes the destructor, but it shall not be used as the unqualified-id in a qualified-id. [Note: A typedef-name that names a class is a class-name (12.1). — end note]

3 The nested-name-specifier :: names the global namespace. A nested-name-specifier that names a namespace (10.3), optionally followed by the keyword template (17.2), 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 or a variable and a prvalue otherwise.

4 A nested-name-specifier that denotes an enumeration (10.2), 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 shall denote the same type in both the context in which the entire qualified-id occurs and in the context of the class denoted by the nested-name-specifier.

8.4.5 Lambda expressions

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 expressions provide a concise way to create simple function objects. [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); });
}
—end example]

A lambda-expression is a prvalue whose result object is called the closure object. [Note: A closure object behaves like a function object (23.14). — end note]

In the decl-specifier-seq of the lambda-declarator, each declSpecifier shall either be mutable or constexpr. [Note: The trailing requires-clause is described in Clause 11. — 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 10.1.7.4. [Example:
    auto x1 = [](int i){ return i; }; // OK: return type is int
    auto x2 = [](int i, int j){ return {i, j}; }; // error: deducing return type from braced-init-list
    auto x3 = ()=auto& [ return j; ]; // OK: return type is int&
—end example]

A lambda is a generic lambda if the auto type-specifier appears as one of the decl-specifiers in the decl-specifier-seq of a parameter-declaration of the lambda-expression, or if the lambda has a template-parameter-list. [Example:
    int i = [](int i, auto a) { return i; }(3, a); // OK: a generic lambda
    int j = []<class T>(T t, int i) { return i; }(3, 4); // OK: a generic lambda
—end example]

8.4.5.1 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 (11.6.1). 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:

— the size and/or alignment of the closure type,
— whether the closure type is trivially copyable (Clause 12), or
— whether the closure type is a standard-layout class (Clause 12).
An implementation shall not add members of rvalue reference type to the closure type.

The closure type for a non-generic lambda-expression has a public inline function call operator (16.5.4) whose parameters and return type are described by the lambda-expression’s parameter-declaration-clause and trailing-return-type respectively. For a generic lambda, the closure type has a public inline function call operator member template (17.6.2) whose template-parameter-list consists of the specified template-parameter-list, if any, to which is appended one invented type template-parameter for each occurrence of auto in the lambda’s parameter-declaration-clause, in order of appearance. The invented type template-parameter is a parameter pack if the corresponding parameter-declaration declares a function parameter pack (11.3.5). The return type and function parameters of the function call operator template are derived from the lambda-expression’s trailing-return-type and parameter-declaration-clause by replacing each occurrence of auto in the decl-specifiers of the parameter-declaration-clause with the name of the corresponding invented template-parameter. The requires-clause of the function call operator template is the requires-clause immediately following < template-parameter-list >, if any. The trailing requires-clause of the function call operator or operator template is the requires-clause following the lambda-declarator, if any. [Example:

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

```cpp
auto vglambda = [](auto printer) {
    return [=](auto&& ... ts) {
        return [=]() {
            printer(ts ...);
        };
    };

    auto p = vglambda( [](auto v1, auto v2, auto v3) {
        std::cout << v1 << v2 << v3;
    });

    auto q = p(1, 'a', 3.14); // OK: outputs 1a3.14
    q(); // OK: outputs 1a3.14
} // end example]
```

The function call operator or operator template is declared const (12.2.2) 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 it satisfies the requirements for a constexpr function (10.1.5). [Note: Names referenced in the lambda-declarator are looked up in the context in which the lambda-expression appears. — end note] [Example:

```cpp
auto ID = [](auto a) { return a; }; // OK
std::assert(ID(3) == 3);
```

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

[Example:

```cpp
auto monoid = [](auto v) { return [=] { return v; }; }; // OK
auto add = [](auto m1) constexpr {
    auto ret = m1();
    return [=](auto m2) mutable {
        auto m1val = m1();
        auto plus = [=](auto m2val) mutable constexpr {
            return m1val += m2val;
        };
        ret = plus(m2());
    };
    ```
return monoid(ret);
};
constexpr auto zero = monoid(0);
constexpr auto one = monoid(1);
static_assert(add(one)(zero)() == one()); // OK

// Since two below is not declared constexpr, an evaluation of its constexpr member function call operator
// cannot perform an lvalue-to-rvalue conversion on one of its subobjects (that represents its capture)
// in a constant expression.
auto two = monoid(2);
assert(two() == 2); // OK, not a constant expression.
static_assert(add(one)(one)() == two()); // ill-formed: two() is not a constant expression
static_assert(add(one)(one)() == monoid(2)()); // OK
—end example

The function call operator or operator template may be constrained (17.4.2) by a constrained-parameter (17.1),
a requires-clause (Clause 17), or a trailing requires-clause (Clause 11). [Example:

template <typename T> concept C1 = /* ... */;
template <std::size_t N> concept C2 = /* ... */;
template <typename A, typename B> concept C3 = /* ... */;

auto f = []<typename T1, C1 T2> requires C2<sizeof(T1) + sizeof(T2)>
(T1 a1, T1 b1, T2 a2, auto a3, auto a4) requires C3<decltype(a4), T2> {
    // T2 is a constrained parameter,
    // 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]

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 (10.5) 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. F is a constexpr function if the function call operator is a
constexpr function. For a generic lambda with no lambda-capture, the closure type has a conversion function
template to pointer to function. The conversion function template has the same invented template parameter
list, and the pointer to function has the same parameter types, as the function call operator template. The
return type of the pointer to function shall behave as if it were a decltype-specifier denoting the return type
of the corresponding function call operator template specialization.

[ Note: If the generic lambda has no trailing-return-type or the trailing-return-type contains a placeholder
type, return type deduction of the corresponding function call operator template specialization has to be done.
The corresponding specialization is that instantiation of the function call operator template with the same
template arguments as those deduced for the conversion function template. Consider the following:

auto glambda = [](auto a) { return a; };
int (*fp)(int) = glambda;

The behavior of the conversion function of glambda above is like that of the following conversion function:

struct Closure {
    template<class T> auto operator()(T t) const { ... }
    template<class T> static auto lambda_call_operator_invoker(T a) {
        // forwards execution to operator()(a) and therefore has
        // the same return type deduced
        ... 
    }
    template<class T> using fptr_t = decltype(lambda_call_operator_invoker(declval<T>())*) (*)(T);
};

§ 8.4.5.1
template<class T> operator fptr_t<T>() const
    { return &lambda_call_operator_invoker; }

—end note

[Example:

    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

—end example]

9 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. F is a constexpr function if the corresponding specialization is a constexpr
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:

    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]

10 The conversion function or conversion function template is public, constexpr, non-virtual, non-explicit, const,
and has a non-throwing exception specification (18.4). [Example:

    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]

11 The lambda-expression’s compound-statement yields the function-body (11.4) of the function call operator,
but for purposes of name lookup (6.4), determining the type and value of this (12.2.2.1) and transforming id-
expressions referring to non-static class members into class member access expressions using (*this) (12.2.2),
the compound-statement is considered in the context of the lambda-expression. [Example:

    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*
        };
    }
    };

§ 8.4.5.1 90
Further, a variable `__func__` is implicitly defined at the beginning of the compound-statement of the lambda-expression, with semantics as described in 11.4.1.

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 (15.8). It has a deleted copy assignment operator if the lambda-expression has a lambda-capture and defaulted copy and move assignment operators otherwise. [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 (15.4).

A member of a closure type shall not be explicitly instantiated (17.8.2), explicitly specialized (17.8.3), or named in a friend declaration (14.3).

### 8.4.5.2 Captures

```cpp
lambda-capture:
    capture-default
    capture-list
    capture-default, capture-list

capture-default:
    &
    =

capture-list:
    capture ..., opt
    capture-list, capture ..., opt

capture:
    simple-capture
    init-capture

simple-capture:
    identifier
    & identifier
    this
    * this

init-capture:
    identifier initializer
    & identifier initializer
```

1. 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.

2. If a lambda-capture includes a capture-default that is `&`, no identifier in a simple-capture of that lambda-capture shall be preceded by `&`. If a lambda-capture includes a capture-default that is `=`, each simple-capture of that lambda-capture shall be of the form “`& identifier`”, “`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
}
```

3. A lambda-expression is a local lambda expression if its innermost enclosing scope is a block scope (6.3.3), or if it appears within a default member initializer and its innermost enclosing scope is the corresponding class scope (6.3.7); any other lambda-expression shall not have a capture-default or simple-capture in its lambda-introducer.
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:

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

An init-capture 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:

- 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
- 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:

```c
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:

- 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 (8.5.2.1) potentially references *this. [Note: This occurs even if overload resolution selects a static member function for the id-expression. —end note]

- A this expression potentially references *this.

- 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 (8.5.1.8) were ignored, the entity is said to be implicitly captured by each intervening lambda-expression with an associated capture-default that does not explicitly capture it. [Example:

```c
void f(int, const int (&)[2] = {});  // #1
void f(const int&, const int (&)[1]);  // #2
void test() {
  const int x = 17;
  auto g = [](auto a) {
    f(x);  // OK: calls #1, does not capture x
  };

  auto g1 = [=](auto a) {
    f(x);  // OK: calls #1, captures x
  };

  auto g2 = [=](auto a) {
    int selector[sizeof(a) == 1 ? 1 : 2]{};
    f(x, selector);  // OK: captures x, might call #1 or #2
  };

  auto g3 = [=](auto a) {
    typeid(a + x);  // captures x regardless of whether a + x is an unevaluated operand
  }
`
Within g1, an implementation might optimize away the capture of \( x \) as it is not odr-used. —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 (9.4.1).] 

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 \( \text{sizeof(int)} \)
        }
    }(0);
}
—end example
—end note
```

§ 8.4.5.2
A lambda-expression appearing in a default argument shall not implicitly or explicitly capture any entity.

[Example:

```c
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
}
@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. However, such an id-expression can still cause the implicit capture of the entity. —end note] If *this is captured by copy, each expression that odr-uses *this is transformed to instead refer to the corresponding unnamed data member of the closure type. [Example:

```c
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:

```c
// 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:

```c
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:
© ISO/IEC N4713

(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;
    
    — end example ]
```

When the lambda-expression is evaluated, the entities that are captured by copy are used to direct-initialize each corresponding non-static data member of the resulting closure object, and the non-static data members corresponding to the init-captures are initialized as indicated by the corresponding initializer (which may be copy- or direct-initialization). (For array members, the array elements are direct-initialized in increasing subscript order.) These initializations are performed in the (unspecified) order in which the non-static data members are declared. [Note: This ensures that the destructions will occur in the reverse order of the constructions. — end note]

[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 (17.6.3). An init-capture followed by an ellipsis is ill-formed. [Example:

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

— end example ]

8.4.6 Fold expressions [expr.prim.fold]

A fold expression performs a fold of a template parameter pack (17.6.3) 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 parameter pack (17.6.3).

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 parameter pack or \( e2 \) shall contain an unexpanded parameter pack, but not both. If \( e2 \) contains an unexpanded parameter pack, the expression is called a binary left fold. If \( e1 \) contains an unexpanded parameter pack, the expression is called a binary right fold. [Example:
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 parameter packs
}

— end example —

8.4.7 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

2 A requires-expression is a prvalue of type bool whose value is described below. Expressions appearing within a requirement-body are unevaluated operands (8.2).

3 [Example: A common use of requires-expressions is to define requirements in concepts such as the one below:

    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 17) as a way of writing ad hoc constraints on template arguments such as the one below:

    template<typename T>
    requires requires (T x) { x + x; }
    T add(T a, T b) { return a + b; }

    The first requires introduces the requires-clause, and the second introduces the requires-expression. — end example]

4 A requires-expression may introduce local parameters using a parameter-declaration-clause (11.3.5). 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:

    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
    };
— end example]

8.4.7.1 Simple requirements [expr.prim.req.simple]

simple-requirement:
expression ;

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 (8.2). — 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]

8.4.7.2 Type requirements [expr.prim.req.type]

type-requirement:
    typename nested-name-specifier_opt type-name ;

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>;
    typename Ref<T>; // required class template specialization,
                        // 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).

8.4.7.3 Compound requirements [expr.prim.req.compound]

compound-requirement:
    {
        expression } noexcept_opt return-type-requirement_opt ;

    return-type-requirement:
        trailing-return-type
        -> cv-qualifier-seq_opt constrained-parameter cv-qualifier-seq_opt abstract-declarator_opt

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 (18.4).
If the return-type-requirement is present, then:

— Substitution of template arguments (if any) into the return-type-requirement is performed.

— If the return-type-requirement is a trailing-return-type, E is implicitly convertible to the type named by the trailing-return-type. If conversion fails, the enclosing requires-expression is false.

— If the return-type-requirement starts with a constrained-parameter (17.1), the expression is deduced against an invented function template F using the rules in 17.9.2.1. F is a void function template with a single type template parameter T declared with the constrained-parameter. A cv-qualifier-seq cv is formed as the union of const and volatile specifiers around the constrained-parameter. F has a single parameter whose type-specifier is cv T followed by the abstract-declarator. If deduction fails, the enclosing requires-expression is false.

Example:

```cpp
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++;

```cpp
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.

```cpp
template<typename T, typename U> concept C3 = requires (T t, U u) {
   t == u;
};
template<typename T> concept C4 = requires(T x) {
   {*x} -> C3<int> const&;
};
```

The compound-requirement requires that *x be deduced as an argument for the invented function:

```cpp
template<int> X void f(X const&);
```

In this case, deduction only succeeds if an expression of the type deduced for X can be compared to an int with the == operator.

```cpp
template<typename T> concept C5 = requires(T x) {
   {g(x)} noexcept;
};
```

The compound-requirement in C5 requires that g(x) is a valid expression and that g(x) is non-throwing.

---

8.4.7.4 Nested requirements

A nested-requirement can be used to specify additional constraints in terms of local parameters. The constraint-expression shall be satisfied (17.4.2) 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 17.4.2.

Example:

```cpp
template<typename U> concept C = sizeof(U) == 1;

template<typename T> concept D = requires (T t) {
   requires C<decltype (+t)>
};
```

D<T> is satisfied if sizeof(decltype (+t)) == 1 (17.4.1.2).

---

1 A local parameter shall only appear as an unevaluated operand (8.2) 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]

8.5 Compound expressions

8.5.1 Postfix expressions

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 . pseudo-destructor-name
  postfix-expression -> pseudo-destructor-name
  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

pseudo-destructor-name:
  nested-name-specifier_opt type-name :: ~ type-name
  nested-name-specifier template simple-template-id :: ~ type-name
  ~ type-name
  ~ decltype-specifier

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 (17.2). — end note ]

8.5.1.1 Subscripting

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.66 The expression E1[E2] is identical (by definition) to *(E1)+(E2)) [ Note: see 8.5.2 and 8.5.6 for details of * and + and 11.3.4 for details of arrays. — end note ] , 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 A braced-init-list shall not be used with the built-in subscript operator.

8.5.1.2 Function call

A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of initializer-clauses which constitute the arguments to the function. 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 be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (7.3) is suppressed on the postfix expression), or it shall have function pointer type. 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 (10.5). For a call to a non-static member function,
the postfix expression shall be an implicit (12.2.2, 12.2.3) or explicit class member access (8.5.1.5) whose
id-expression is a function member name, or a pointer-to-member expression (8.5.4) 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 12.2.2). — end note] If a function or member function
name is used, the name can be overloaded (Clause 16), in which case the appropriate function shall
be selected according to the rules in 16.3. 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 (13.3)
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.
15.7 describes the behavior of virtual function calls when the object expression refers to an object under
construction or destruction. — end note]

2 [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]

3 If the postfix-expression designates a destructor (15.4), the type of the function call expression is void;
otherwise, the type of the function call expression is the return type of the statically chosen function (i.e.,
ignoring the virtual keyword), even if the type of the function actually called is different. This return type
shall be an object type, a reference type or cv void.

4 When a function is called, each parameter (11.3.5) shall be initialized (11.6, 15.8, 15.1) with its corresponding
argument. If the function is a non-static member function, the this parameter of the function (12.2.2.1) shall
be initialized with a pointer to the object of the call, converted as if by an explicit type conversion (8.5.3).
[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 13.2, 14.2, and 8.5.1.5. — end note]
When a function is called, the parameters that have object type shall have completely-defined object
type. [Note: this still allows a parameter to be a pointer or reference to an incomplete class type. However,
it prevents a passed-by-value parameter to have an incomplete 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 18) with a handler that could handle the exception, this
handler is not considered. — end example]

5 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 16.3.1.2. — 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 8.5.7), but it is unspecified whether the value of j
is 1 or 2. — end example]

6 The result of a function call is the result of the operand of the evaluated return statement (9.6.3) in the
called function (if any), except in a virtual function call if the return type of the final overrider is different

§ 8.5.1.2
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 (11.3.2): 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 (10.1.7, 5.13, 5.13.5, 11.3.4, 15.2). 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 (11.3.6)) or more arguments (by using the ellipsis, . . . , or a function parameter pack (11.3.5)) than the number of parameters in the function definition (11.4). [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 (21.11). [Note: This paragraph does not apply to arguments passed to a function parameter pack. Function parameter packs are expanded during template instantiation (17.6.3), thus each such argument has a corresponding parameter when a function template specialization is actually called. —end note—}

The only effect is the evaluation of the value of the argument is converted to the promoted type before the call. 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 12) 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 (7.6), or a floating-point type that is subject to the floating-point promotion (7.7), 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 xvalue if the result type is an rvalue reference to object type, and a prvalue otherwise.

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

A simple-type-specifier (10.1.7.2) or typename-specifier (17.7) 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 (16.3.1.8) 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 (8.5.3). Otherwise, if the type is cv void and the initializer is (), 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 (11.6) with the initializer. For an expression of the form T(), T shall not be an array type.

8.5.1.4 Pseudo destructor call [expr.pseudo]

The use of a pseudo-destructor-name after a dot . or arrow -> operator represents the destructor for the non-class type denoted by type-name or decltype-specifier. The result shall only be used as the operand for the function call operator (), and the result of such a call has type void. The only effect is the evaluation of the postfix-expression before the dot or arrow.

The left-hand side of the dot operator shall be of scalar type. The left-hand side of the arrow operator shall be of pointer to scalar type. This scalar type is the object type. The cv-unqualified versions of the object type and of the type designated by the pseudo-destructor-name shall be the same type. Furthermore, the two type-names in a pseudo-destructor-name of the form

nested-name-specifier, type-name :: ~ type-name

shall designate the same scalar type (ignoring cv-qualification).

8.5.1.5 Class member access [expr.ref]

A postfix expression followed by a dot . or an arrow ->, optionally followed by the keyword template (17.2), and then followed by an id-expression, is a postfix expression. The postfix expression before the dot or arrow
is evaluated, the result of that evaluation, together with the id-expression, determines the result of the entire postfix expression.

2 For the first option (dot) the first expression shall be a glvalue having class type. For the second option (arrow) the first expression shall be a prvalue having pointer to class type. In both cases, 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 expression E1->E2 is converted to the equivalent form *(E1)->E2; the remainder of 8.5.1.5 will address only the first option (dot). In either case, 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 12), 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]

3 Abbreviating postfix-expression.id-expression as E1.E2, E1 is called the object expression. If E2 is a bit-field, E1.E2 is a bit-field. The type and value category of E1.E2 are determined as follows. In the remainder of 8.5.1.5, cq represents either const or the absence of const and vq represents either volatile or the absence of volatile. cv represents an arbitrary set of cv-qualifiers, as defined in 6.7.3.

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

(4.1) — If E2 is a static data member and the type of E2 is T, then E1.E2 is an lvalue; the expression designates the named member of the class. The type of E1.E2 is T.

(4.2) — If E2 is a non-static data member and the type of E1 is “cq1 vq1 X”, and the type of E2 is “cq2 vq2 T”, the expression designates the named member of the object designated by the first expression. If E1 is an lvalue, then E1.E2 is an lvalue; otherwise E1.E2 is an xvalue. Let the notation vq12 stand for the “union” of vq1 and vq2; that is, if vq1 or vq2 is volatile, then vq12 is volatile. Similarly, let the notation cq12 stand for the “union” of cq1 and cq2; that is, if cq1 or cq2 is const, then cq12 is const. If E2 is declared to be a mutable member, then the type of E1.E2 is “vq12 T”. If E2 is not declared to be a mutable member, then the type of E1.E2 is “cq12 vq12 T”.

(4.3) — If E2 is a (possibly overloaded) member function, function overload resolution (16.3) is used to determine whether E1.E2 refers to a static or a non-static member function.

(4.3.1) — If it refers to a static member function and the type of E2 is “function of parameter-type-list returning T”, then E1.E2 is an lvalue; the expression designates the static member function. The type of E1.E2 is the same type as that of E2, namely “function of parameter-type-list returning T”.

(4.3.2) — Otherwise, if E1.E2 refers to a non-static member function and the type of E2 is “function of parameter-type-list cv ref-qualifier opt returning T”, then E1.E2 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 (12.2.1). [Note: Any redundant set of parentheses surrounding the expression is ignored (8.4). —end note] The type of E1.E2 is “function of parameter-type-list cv returning T”.

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

(4.5) — If E2 is a member enumerator and the type of E2 is T, the expression E1.E2 is a prvalue. The type of E1.E2 is T.

5 If E2 is a non-static data member or a non-static member function, the program is ill-formed if the class of which E2 is directly a member is an ambiguous base (13.2) of the naming class (14.2) of E2. [Note: The program is also ill-formed if the naming class is an ambiguous base of the class type of the object expression; see 14.2. —end note]

8.5.1.6 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 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-sequenced 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 8.5.6 and 8.5.18.

2 The operand of postfix -- is decremented analogously to the postfix ++ operator. [Note: For prefix increment and decrement, see 8.5.2.2. —end note]

8.5.1.7 Dynamic cast [expr.dynamic.cast]

1 The result of the expression dynamic_cast<T>(v) is the result of converting the expression v to type T. T shall be a pointer or reference to a complete class type, or “pointer to cv void”. The dynamic_cast operator shall not cast away constness (8.5.1.11).

2 If T is a pointer type, v shall be a prvalue of a pointer to complete class type, and the result is a prvalue of type T. If T is an lvalue reference type, v shall be an lvalue of a complete class type, and the result is an lvalue of the type referred to by T. If T is an rvalue reference type, v shall be a glvalue having a complete class type, and the result is an xvalue of the type referred to by T.

3 If the type of v is the same as T, or it is the same as T except that the class object type in T is more cv-qualified than the class object type in v, the result is v (converted if necessary).

4 If the value of v is a null pointer value in the pointer case, the result is the null pointer value of type T.

5 If T is “pointer to cv void” 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. 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. 69 In both the pointer and reference cases, the program is ill-formed if cv2 has greater cv-qualification than cv1 or if B is an inaccessible or ambiguous base class of D. [Example:

```c
struct B { };
struct D : B { };
void foo(D* dp) {
    B* bp = dynamic_cast<B*>(dp);  // equivalent to B* bp = dp;
}
```

—end example]

6 Otherwise, v shall be a pointer to or a glvalue of a polymorphic type (13.3).

7 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.

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

(8.1) — If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class subobject of a C object, and if only one object of type C is derived from the subobject pointed (referred) to by v the result points (refers) to that C object.

(8.2) — Otherwise, if v points (refers) to a public base class subobject of the most derived object, and the type of the most derived object has a base class, of type C, that is unambiguous and public, the result points (refers) to the C subobject of the most derived object.

(8.3) — Otherwise, the runtime check fails.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws an exception (18.1) of a type that would match a handler (18.3) of type std::bad_cast (21.7.3).

[Example:

```c
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
    D d;
    B* bp = (B*)&d;  // cast needed to break protection
}
```

69) 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.
8.5.1.8 Type identification [expr typeid]
1 The result of a typeid expression is an lvalue of static type const std::type_info (21.7.2) 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 21.7.2. 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 expression whose type is a polymorphic class type (13.3), 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 expression is obtained by applying the unary * operator to a pointer and the pointer is a null pointer value (7.11), the typeid expression throws an exception (18.1) of a type that would match a handler of type std::bad_typeid exception (21.7.4).

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 (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) conversions are not applied to the expression. If the expression is a prvalue, the temporary materialization conversion (7.4) is applied. The expression is an unevaluated operand (8.2).

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> (21.7.2) is not included prior to a use of typeid, the program is ill-formed.  
[Note: 15.7 describes the behavior of typeid applied to an object under construction or destruction. —end note]
8.5.1.9 Static cast

The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type or an rvalue reference to function type, the result is an lvalue; if `T` is an rvalue reference to object type, the result is an xvalue; otherwise, the result is a prvalue. The `static_cast` operator shall not cast away constness (8.5.1.11).

An lvalue of type “`cv1 B`”, where `B` is a class type, can be cast to type “reference to `cv2 D`”, where `D` is a class derived (Clause 13) 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 (7.11), 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:
```
struct B { };  
struct D : public B { };  
D d;  
B &br = d;  
static_cast<D&>(br);  // produces lvalue to the original d 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`” (11.6.3). 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 (7.1) 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 (Clause 14) or ambiguous (13.2) base class of `T1`, a program that necessitates such a cast is ill-formed.

An expression `e` can be explicitly converted to a type `T` if there is an implicit conversion sequence (16.3.3.1) from `e` to `T`, or if overload resolution for a direct-initialization (11.6) of an object or reference of type `T` from `e` would find at least one viable function (16.3.2). 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` (11.6) and then using the temporary variable as the result of the conversion. Otherwise, the result object is direct-initialized from `e`. [Note: The conversion is ill-formed when attempting to convert an expression of class type to an inaccessible or ambiguous base class. —end note]

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 (8.2). [Note: However, if the value is in a temporary object (15.2), 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 (Clause 7) not containing an lvalue-to-rvalue (7.1), array-to-pointer (7.2), function-to-pointer (7.3), null pointer (7.11), null member pointer (7.12), boolean (7.14), or function pointer (7.13) 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:
```
struct B { };  
struct D : private B { };  
void f()  
{   
    static_cast<D*>(B@0));  // error: B is a private base of D  
    static_cast<int B::*>(int D::*0);  // error: B is a private base of D  
}  
—end example]
```

The value-to-rvalue (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (8.5.1.11), and the following additional rules for specific cases:
A value of a scoped enumeration type \((10.2)\) can be explicitly converted to an integral type. When that type is \(cv\) bool, the resulting value is \texttt{false} if the original value is zero and \texttt{true} for all other values. For the remaining integral types, the value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified. 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 \((10.2)\), 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 \((7.10)\), 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 class derived (Clause 13) 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 \((7.11)\), the program is ill-formed. The null pointer value \((7.11)\) 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 cv1 T” can be converted to a prvalue of type “pointer to member of \(B\) of type cv2 T”, where \(B\) is a base class (Clause 13) of \(D\), if \(cv2\) is the same cv-qualification as, or greater cv-qualification than, \(cv1\).22 If no valid standard conversion from “pointer to member of \(B\) of type \(T\)” to “pointer to member of \(D\) of type \(T\)” exists \((7.12)\), the program is ill-formed. The null member pointer value \((7.12)\) is converted to the null member pointer value of the destination type. If \(B\) 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 8.5.4. — end note]

A prvalue of type “pointer to cv1 void” can be converted to a prvalue of type “pointer to cv2 T”, where \(T\) is an object type and \(cv2\) is the same cv-qualification as, or greater cv-qualification than, \(cv1\). If the original pointer value represents the address \(A\) of a byte in memory and \(A\) does not satisfy the alignment requirement of \(T\), then the resulting pointer value is unspecified. Otherwise, if the original pointer value points to an object \(a\), and there is an object \(b\) of type \(T\) (ignoring cv-qualification) that is pointer-interconvertible \((6.7.2)\) 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]

8.5.1.10 Reinterpret cast 

The result of the expression \texttt{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 \((7.1)\), array-to-pointer \((7.2)\), and function-to-pointer \((7.3)\) standard conversions are performed on the expression \(v\). Conversions that can be performed explicitly using \texttt{reinterpret_cast} are listed below. No other conversion can be performed explicitly using \texttt{reinterpret_cast}.

The \texttt{reinterpret_cast} operator shall not cast away constness \((8.5.1.11)\). An expression of integral, enumeration, pointer, or pointer-to-member type can be explicitly converted to its own type; such a cast yields the value of its operand.

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

A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [Note: It is intended to be unsurprising to those who know the addressing structure

---

22 Function types (including those used in pointer-to-member-function types) are never cv-qualified; see 11.3.5.
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]

5 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 6.6.4.4.3, the result of such a conversion will not be a safely-derived pointer value. —end note]

6 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 (11.3.5) 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 7.11 for more details of pointer conversions. —end note]

7 An object pointer can be explicitly converted to an object pointer of a different type. When a prvalue v of object pointer type is converted to the object pointer type “pointer to cv T”, the result is `static_cast<cv T*>(v);` [ Note: Converting a prvalue of type “pointer to T1” to the type “pointer to T2” (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value. —end note]

8 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.

9 The null pointer value (7.11) 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]

10 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 (7.12) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

(10.1) — 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.

(10.2) — 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.

11 A glvalue expression 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 (15.1) or conversion functions (15.3) are called. [Note: Subject to the restrictions in this subclause, an expression may be cast to its own type using a `const_cast` operator. —end note]

8.5.1.11 Const cast

The result of the expression `static_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 (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) 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]

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

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

75) This is sometimes referred to as a type pun when the result refers to the same object as the source glvalue.
For two similar types \( T_1 \) and \( T_2 \) (7.5), a prvalue of type \( T_1 \) may be explicitly converted to the type \( T_2 \) 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{}; // OK, reference binds to temporary array object after qualification conversion to type CA
A &&r1 = const_cast<A>(CA{}); // error: temporary array decayed to pointer
A &&r2 = const_cast<A&&>(CA{}); // OK
```

—end example]

For two object types \( T_1 \) and \( T_2 \), if a pointer to \( T_1 \) can be explicitly converted to the type “pointer to \( T_2 \)” using a `const_cast`, then the following conversions can also be made:

- (4.1) — an lvalue of type \( T_1 \) can be explicitly converted to an lvalue of type \( T_2 \) using the cast `const_cast<T2&>`;
- (4.2) — a glvalue of type \( T_1 \) can be explicitly converted to an xvalue of type \( T_2 \) using the cast `const_cast<T2&&>`; and
- (4.3) — if \( T_1 \) is a class type, a prvalue of type \( T_1 \) can be explicitly converted to an xvalue of type \( T_2 \) using the cast `const_cast<T2&&>`.

The result of a reference `const_cast` refers to the original object if the operand is a glvalue and to the result of applying the temporary materialization conversion (7.4) otherwise.

A null pointer value (7.11) is converted to the null pointer value of the destination type. The null member pointer value (7.12) is converted to the null member pointer value of the destination type.

[Note: Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a `const_cast` that casts away a const-qualifier may produce undefined behavior (10.1.7.1). —end note]

A conversion from a type \( T_1 \) to a type \( T_2 \) casts away constness if \( T_1 \) and \( T_2 \) are different, there is a cv-decomposition (7.5) of \( T_1 \) yielding \( n \) such that \( T_2 \) has a cv-decomposition of the form

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

and there is no qualification conversion that converts \( T_1 \) to

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

Casting from an lvalue of type \( T_1 \) to an lvalue of type \( T_2 \) using an lvalue reference cast or casting from an expression of type \( T_1 \) to an xvalue of type \( T_2 \) using an rvalue reference cast casts away constness if a cast from a prvalue of type “pointer to \( T_1 \)” to the type “pointer to \( T_2 \)” casts away constness.

[Note: Some conversions which involve only changes in cv-qualification cannot be done using `const_cast`. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. —end note]

### 8.5.2 Unary expressions

Expressions with unary operators group right-to-left.

```
unary-expression:
  postfix-expression
  ++ cast-expression
  -- cast-expression
  unary-operator cast-expression
  sizeof unary-expression
  sizeof ( type-id )
  sizeof ... ( identifier )
  alignof ( type-id )
  noexcept-expression
  new-expression
  delete-expression
```

76) `const_cast` is not limited to conversions that cast away a const-qualifier.
8.5.2.1 Unary operators

1 The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is "pointer to T", the type of the result is "T". [Note: Indirection through a pointer to an incomplete type (other than cv void) is valid. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to a prvalue, see 7.1. — end note]

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

3 The result of the unary & operator is a pointer to its operand. The operand shall be an lvalue or a qualified-id. If the operand is a qualified-id naming a non-static or variant member m of some class C with type T, the result has type “pointer to member of class C of type T” and is a prvalue designating C::m. Otherwise, if the type of the expression is T, the result has type “pointer to 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 "cv T" is “pointer to cv T", with the same cv-qualification. — end note] For purposes of pointer arithmetic (8.5.6) and comparison (8.5.9, 8.5.10), 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:]

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

[Note: A pointer to member formed from a mutable non-static data member (10.1.1) 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). 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 16) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 16.4). [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 (Clause 7); 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 class-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 class-name or decltype-specifier in a member access expression or qualified-id is unambiguously parsed as a destructor name.  — end note]

8.5.2.2 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 (8.5.6) and assignment operators (8.5.18) 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 8.5.1.6. — end note]

8.5.2.3 Sizeof [expr.sizeof]
1 The sizeof operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (8.2), 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. sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 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.77 — 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 or 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 size of a most derived class shall be greater than zero (6.6.2). The result of applying sizeof to a base class subobject is the size of the base class type.78 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 sizeof operator can be applied to a pointer to a function, but shall not be applied directly to a function.

4 The lvalue-to-rvalue (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) standard conversions are not applied to the operand of sizeof. If the operand is a prvalue, the temporary materialization conversion (7.4) is applied.

5 The identifier in a sizeof... expression shall name a parameter pack. The sizeof... operator yields the number of arguments provided for the parameter pack identifier. A sizeof... expression is a pack expansion (17.6.3). [Example:

```cpp
template<class... Types>
struct count {
  static const std::size_t value = sizeof...(Types);
};

—end example]
```

6 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> (21.2.1, 21.2.4). — end note]

8.5.2.4 New [expr.new]
1 The new-expression attempts to create an object of the type-id (11.1) 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, 13.4). [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
```

77) sizeof(bool) is not required to be 1.

78) The actual size of a base class subobject may be less than the result of applying sizeof to the subobject, due to virtual base classes and less strict padding requirements on base class subobjects.
new-placement: (expression-list)
new-type-id: type-specifier-seq new-declarator_{opt}
new-declarator: ptr-operator new-declarator_{opt}
noptr-new-declarator
[ expression ] attribute-specifier-seq_{opt}
noptr-new-declarator [ constant-expression ] attribute-specifier-seq_{opt}
new-initializer: (expression-list_{opt}) braced-init-list

Entities created by a new-expression have dynamic storage duration (6.6.4.4). [ Note: The lifetime of such an entity is not necessarily restricted to the scope in which it is created. — end note ] If the entity is a non-array object, the result of the new-expression is a pointer to the object created. If it is an array, the result of the new-expression is a pointer to the initial element of the array.

2 If a placeholder type (10.1.7.4) appears in the type-specifier-seq of a new-type-id or type-id of a new-expression, the allocated type is deduced as follows: Let init be the new-initializer, if any, and T be the new-type-id or type-id of the new-expression, then the allocated type is the type deduced for the variable x in the invented declaration (10.1.7.4):

T x init ;

[ Example:
new auto(1); // allocated type is int
auto x = new auto(‘a’); // allocated type is char, x is of type char *

template<class T> struct A { A(T, T); }; 
auto y = new A(1, 2); // allocated type is A<int>
— end example ]

3 The new-type-id in a new-expression is the longest possible sequence of new-declarators. [ Note: This prevents ambiguities between the declarator operators & , &&, *, and [] and their expression counterparts. — end note ] [ Example:
new int * i; // syntax error: parsed as (new int*) i, not as (new int)*i
The * is the pointer declarator and not the multiplication operator. — end example ]

4 [ Note: Parentheses in a new-type-id of a new-expression can have surprising effects. [ Example:
new int(*[10])(); // error
is ill-formed because the binding is
(new int) (*[10])(); // error
Instead, the explicitly parenthesized version of the new operator can be used to create objects of compound types (6.7.2):
new (int (*[10]))();
allocates an array of 10 pointers to functions (taking no argument and returning int). — end example ] — end note]

5 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.

6 Every constant-expression in a noptr-new-declarator shall be a converted constant expression (8.6) of type std::size_t and shall evaluate to a strictly positive value. The expression in a noptr-new-declarator is implicitly converted to std::size_t. [ 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 ]

§ 8.5.2.4 111
The expression in a `nptr-new-declarator` is erroneous if:

1. the expression is of non-class type and its value before converting to `std::size_t` is less than zero;
2. the expression is of class type and its value before application of the second standard conversion (16.3.1.2)\(^7\) is less than zero;
3. its value is such that the size of the allocated object would exceed the implementation-defined limit (Annex B); or
4. the `new-initializer` is a `braced-init-list` and the number of array elements for which initializers are provided (including the terminating ‘\0’ in a string literal (5.13.5)) exceeds the number of elements to initialize.

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

1. if the expression is a core constant expression, the program is ill-formed;
2. otherwise, an allocation function is not called; instead
   - if the allocation function that would have been called has a non-throwing exception specification (18.4), the value of the `new-expression` is the null pointer value of the required result type;
   - otherwise, the `new-expression` terminates by throwing an exception of a type that would match a handler (18.3) of type `std::bad_array_new_length` (21.6.3.2).\(^8\)

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

A `new-expression` may obtain storage for the object by calling an allocation function (6.6.4.4.1). If the `new-expression` terminates by throwing an exception, it may release storage by calling a deallocation function (6.6.4.4.2). 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[]`.\[^{21.6.2.1, 21.6.2.2}^\]

An implementation is allowed to omit a call to a replaceable global allocation function (21.6.2.1, 21.6.2.2). 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:

1. the evaluation of `e1` is sequenced before the evaluation of `e2`, and
2. `e2` is evaluated whenever `e1` obtains storage, and
3. both `e1` and `e2` invoke the same replaceable global allocation function, and
4. if the allocation function invoked by `e1` and `e2` is throwing, any exceptions thrown in the evaluation of either `e1` or `e2` would be first caught in the same handler, and
5. the pointer values produced by `e1` and `e2` are operands to evaluated `delete-expression`s, and
6. the evaluation of `e2` is sequenced before the evaluation of the `delete-expression` whose operand is the pointer value produced by `e1`.

\[^{Example:}\]

```cpp
void 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]};
}
```

\(^{79}\) 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.
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;
    }
}

— end example]

11 When a new-expression calls an allocation function and that allocation has not been extended, the new-expression passes the amount of space requested to the allocation function as the first argument of type std::size_t. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array. For arrays of char, unsigned char, and std::byte, the difference between the result of the new-expression and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (6.6.5) 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 ]

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

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

14 Overload resolution is performed on a function call created by assembling an argument list. The first argument is the amount of space requested, and has type std::size_t. If the type of the allocated object has new-extended alignment, the next argument is the type's alignment, and has type std::align_val_t. If the new-placement syntax is used, the initializer-clauses in its expression-list are the succeeding arguments. If no matching function is found and the allocated object type has new-extended alignment, the alignment argument is removed from the argument list, and overload resolution is performed again.

15 [ Example:

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

(15.2) — new(2,f) T results in one of the following calls:
    operator new(sizeof(T), 2, f)
    operator new(sizeof(T), std::align_val_t(alignof(T)), 2, f)

(15.3) — 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)))

(15.4) — 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]
Note: Unless an allocation function has a non-throwing exception specification (18.4), it indicates failure to allocate storage by throwing a std::bad_alloc exception (6.6.4.4.1, Clause 18, 21.6.3.1); 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 (21.6.2.3) 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.

When the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. — end note]

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

— If the new-initializer is omitted, the object is default-initialized (11.6). [ Note: If no initialization is performed, the object has an indeterminate value. — end note]

— Otherwise, the new-initializer is interpreted according to the initialization rules of 11.6 for direct-initialization.

The invocation of the allocation function is sequenced before the evaluations of expressions in the new-initializer. Initialization of the allocated object is sequenced before the value computation of the new-expression.

If the new-expression creates an object or an array of objects of class type, access and ambiguity control are done for the allocation function, the deallocation function (15.5), and the constructor (15.1). If the new-expression creates an array of objects of class type, the destructor is potentially invoked (15.4).

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]

If the new-expression begins with a unary :: operator, the deallocation function’s name is looked up in the global scope. Otherwise, if the allocated type is a class type T or an array thereof, the deallocation function’s name is looked up in the scope of T. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function’s name is looked up in the global scope.

A declaration of a placement deallocation function matches the declaration of a placement allocation function if it has the same number of parameters and, after parameter transformations (11.3.5), 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 (6.6.4.4.2) 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 a non-placement allocation function, the normal deallocation function lookup is used to find the matching deallocation function (8.5.2.5) [ Example:

```cpp
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]
```

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

80) This may include evaluating a new-initializer and/or calling a constructor.
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.

### 8.5.2.5 Delete

1 The *delete-expression* operator destroys a most derived object (6.6.2) or array created by a *new-expression*.  

```
.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.\(^{81}\) The operand shall be of pointer to object type or of class type. If of class type, the operand is contextually implicitly converted (Clause 7) to a pointer to object type.\(^{82}\) 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 (Clause 13). 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`.\(^{83}\) 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 (8.5.1.11) of the pointer expression before it is used as the operand of the `delete-expression`. — end note]

3 In a single-object delete expression, if the static type of the object to be deleted is different from its dynamic type, 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, the *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 15.6.2).

7 If the value of the operand of the *delete-expression* is not a null pointer value, then:

   - If the allocation call for the *new-expression* for the object to be deleted was not omitted and the allocation was not extended (8.5.2.4), the *delete-expression* shall call a deallocation function (6.6.4.4.2). The value returned from the allocation call of the *new-expression* shall be passed as the first argument to the deallocation function.

   - 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.

   - 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.

---

\(^{81}\) 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.

\(^{82}\) This implies that an object cannot be deleted using a pointer of type `void*` because `void` is not an object type.

\(^{83}\) 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.
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 (15.5). If no class-specific deallocation function is found, the deallocation function's name is looked up in global scope.

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

1. If 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 exactly one preferred function is found, that function is selected and the selection process terminates. If more than one preferred function is found, all non-preferred functions are eliminated from further consideration.
2. If the deallocation functions have class scope, the one without a parameter of type `std::size_t` is selected.
3. If the type is complete and if, for the second alternative (delete array) 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.
4. Otherwise, it is unspecified whether a deallocation function with a parameter of type `std::size_t` is selected.

When a `delete-expression` is executed, the selected deallocation function shall be called with the address of the most-derived object in a single-object delete expression, or the address of the object suitably adjusted for the array allocation overhead (8.5.2.4) in an array delete expression, as its first argument. If a deallocation function with a parameter of type `std::align_val_t` is used, the alignment of the type of the object to be deleted is passed as the corresponding argument. If a deallocation function with a parameter of type `std::size_t` is used, the size of the most-derived type, or of the array plus allocation overhead, respectively, is passed as the corresponding argument.84 [Note: If this results in a call to a usual deallocation function, and either the first argument was not the result of a prior call to a usual allocation function or the second argument was not the corresponding argument in said call, the behavior is undefined (21.6.2.1, 21.6.2.2). —end note]

Access and ambiguity control are done for both the deallocation function and the destructor (15.4, 15.5).

### 8.5.2.6 Alignof

An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a `type-id` representing a complete object type, or an array thereof, or a reference to one of those types.

The result is an integral constant of type `std::size_t`.

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.

### 8.5.2.7 noexcept operator

The `noexcept` operator determines whether the evaluation of its operand, which is an unevaluated operand (8.2), can throw an exception (18.1).

```c
noexcept-expression:  
  noexcept ( expression )
```

The result of the `noexcept` operator is a constant of type `bool` and is a prvalue.

The result of the `noexcept` operator is `true` unless the `expression` is potentially-throwing (18.4).

### 8.5.3 Explicit type conversion (cast notation)

The result of the expression `(T) cast-expression` is of type `T`. The result is an lvalue if `T` is an lvalue reference type or an rvalue reference to function type and an xvalue if `T` is an rvalue reference to object type; otherwise the result is a prvalue. [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 8.2. —end note]

84) If the static type of the object to be deleted is complete and is different from the dynamic type, and the destructor is not virtual, the size might be incorrect, but that case is already undefined, as stated above.
An explicit type conversion can be expressed using functional notation (8.5.1.3), a type conversion operator (\texttt{dynamic_cast}, \texttt{static_cast}, \texttt{reinterpret_cast}, \texttt{const_cast}), or the \texttt{cast} notation.

\texttt{cast-expression}:

\verb|unary-expression| \texttt{cast-expression}

Any type conversion not mentioned below and not explicitly defined by the user (15.3) is ill-formed.

The conversions performed by

1. a \texttt{const_cast} (8.5.1.11),
2. a \texttt{static_cast} (8.5.1.9),
3. a \texttt{static_cast} followed by a \texttt{const_cast},
4. a \texttt{reinterpret_cast} (8.5.1.10), or
5. a \texttt{reinterpret_cast} followed by a \texttt{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 \texttt{static_cast} in the following situations the conversion is valid even if the base class is inaccessible:

1. 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;
2. 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;
3. 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 \texttt{static_cast} followed by a \texttt{const_cast}, the conversion is ill-formed.

\[\text{Example:}\]

\begin{verbatim}
struct A { };
struct I1 : A { };
struct I2 : A { };
struct D : I1, I2 { };
A* foo( D* p ) {
  return (A*)( p ); // ill-formed \texttt{static_cast} interpretation
}
\end{verbatim}

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 \texttt{static_cast} or the \texttt{reinterpret_cast} interpretation is used, even if there is an inheritance relationship between the two classes. \[\text{Note: For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted to interpret a cast between pointers to the classes as if the class types were complete at the point of the cast. — end note}\]

8.5.4 Pointer-to-member operators [expr.mptr.oper]

The pointer-to-member operators \texttt{->*} and \texttt{.*} group left-to-right.

\texttt{pm-expression}:

\verb|cast-expression| \texttt{pm-expression} \texttt{.*} \texttt{cast-expression} \texttt{pm-expression} \texttt{->*} \texttt{cast-expression}

The binary operator \texttt{.*} binds its second operand, which shall be of type “pointer to member of \texttt{T}” to its first operand, which shall be a glvalue of class \texttt{T} or of a class of which \texttt{T} is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.
The binary operator \( \rightarrow^{*} \) binds its second operand, which shall be of type “pointer to member of \( T \)” to its first operand, which shall be of type “pointer to \( U \)” where \( U \) is either \( T \) or a class of which \( T \) is an unambiguous and accessible base class. The expression \( E_{1} \rightarrow^{*} E_{2} \) is converted into the equivalent form \( \left( \ast (E_{1}) \right) \ast E_{2} \).

Abbreviating \( pm-expression \ast cast-expression \) as \( E_{1} \ast E_{2} \), \( E_{1} \) is called the object expression. If the dynamic type of \( E_{1} \) does not contain the member to which \( E_{2} \) refers, the behavior is undefined. Otherwise, the expression \( E_{1} \) is sequenced before the expression \( E_{2} \).

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 \( E_{1}.E_{2} \) given in 8.5.1.5. [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,

```c
struct S {
    S() : i(0) { }
    mutable int i;
};
void f() {
    const S cs;
    int S::* pm = &S::i;  // pm refers to mutable member S::i
    cs.*pm = 88;          // ill-formed: cs is a const object
}
—end note]
```

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

```c
(ptr_to_obj->*ptr_to_mfct)(10);
```

calls the member function denoted by \( ptr\_to\_mfct \) for the object pointed to by \( ptr\_to\_obj \). — end example ]

In a \( \ast \) 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 \( \ast \) 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 \( \ast \) 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 \( \ast \) expression whose second operand is a pointer to a member function is a prvalue. If the second operand is the null member pointer value (7.12), the behavior is undefined.

### 8.5.5 Multiplicative operators [expr.mul]

1. The multiplicative operators \( \ast \), \( / \), and \( \% \) group left-to-right.

```
multiplicative-expression:
    pm-expression
    multiplicative-expression \ast pm-expression
    multiplicative-expression \slash pm-expression
    multiplicative-expression \% pm-expression
```

2. The operands of \( \ast \) and \( \slash \) shall have arithmetic or unscoped enumeration type; the operands of \( \% \) shall have integral or unscoped enumeration type. The usual arithmetic conversions (8.3) are performed on the operands and determine the type of the result.

3. The binary \( \ast \) operator indicates multiplication.

The binary \( \slash \) 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 \( \slash \) or \( \% \) is zero the behavior is undefined. For integral operands the \( \slash \) operator yields the algebraic quotient with any fractional part discarded;\(^{85} \) if the quotient \( a/b \) is representable in the type of the result, \( (a/b)\ast b + a\%b \) is equal to \( a \); otherwise, the behavior of both \( a/b \) and \( a\%b \) is undefined.

### 8.5.6 Additive operators [expr.add]

1. The additive operators \( + \) and \( - \) group left-to-right. The usual arithmetic conversions (8.3) are performed for operands of arithmetic or enumeration type.

\(^{85} \) This is often called truncation towards zero.
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 that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the expression \( P \) points to element \( x[i] \) of an array object \( x \) with \( n \) elements,\(^{86}\) the expressions \( P + J \) and \( J + P \) (where \( J \) has the value \( j \)) point to the (possibly-hypothetical) element \( x[i + j] \) if \( 0 \leq i + j \leq n \); otherwise, the behavior is undefined. Likewise, the expression \( P - J \) points to the (possibly-hypothetical) element \( x[i - j] \) if \( 0 \leq i - j \leq n \); otherwise, the behavior is undefined.

5 When two pointers to elements of the same array object 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 (21.2). If the expressions \( P \) and \( Q \) point, 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 std::ptrdiff_t, the behavior is undefined. — end note]

6 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 (7.5), 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 If the value 0 is added to or subtracted from a null pointer value, the result is a null pointer value. If two null pointer values are subtracted, the result compares equal to the value 0 converted to the type std::ptrdiff_t.

### 8.5.7 Shift operators

[expr.shift]

1 The shift operators << and >> group left-to-right.

shift-expression:
  additive-expression
  shift-expression << additive-expression
  shift-expression >> additive-expression

The operands shall be of integral or unscoped enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand.

2 The value of \( E1 << E2 \) is \( E1 \) left-shifted \( E2 \) bit positions; vacated bits are zero-filled. If \( E1 \) has an unsigned type, the value of the result is \( E1 \times 2^{E2} \), reduced modulo one more than the maximum value representable in the result type. Otherwise, if \( E1 \) has a signed type and non-negative value, and \( E1 \times 2^{E2} \) is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; otherwise, the behavior is undefined.

3 The value of \( E1 >> E2 \) is \( E1 \) right-shifted \( E2 \) bit positions. If \( E1 \) has an unsigned type or if \( E1 \) has a signed type and a non-negative value, the value of the result is the integral part of the quotient of \( E1/2^{E2} \). If \( E1 \) has a signed type and a negative value, the resulting value is implementation-defined.

4 The expression \( E1 \) is sequenced before the expression \( E2 \).

---

86) An object that is not an array element is considered to belong to a single-element array for this purpose; see 8.5.2.1. 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.
8.5.8 Three-way comparison operator [expr.spaceship]

1 The three-way comparison operator groups left-to-right.

   compare-expression:
     shift-expression
     compare-expression <= shift-expression

2 The expression \( p <=> q \) is a prvalue indicating whether \( p \) is less than, equal to, greater than, or incomparable with \( q \).

3 If one of the operands is of type bool and the other is not, the program is ill-formed.

4 If both operands have arithmetic types, the usual arithmetic conversions (8.3) are applied to the operands. Then:

   (4.1) — If a narrowing conversion (11.6.4) is required, other than from an integral type to a floating point type, the program is ill-formed.

   (4.2) — Otherwise, if the operands have integral type, the result is of type \( \text{std::strong\_ordering} \). 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.

   (4.3) — Otherwise, the operands have floating-point type, and the result is of type \( \text{std::partial\_ordering} \). The expression \( a <=> b \) 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 \( <=> \) to the converted operands.

6 If at least one of the operands is of pointer type, array-to-pointer conversions (7.2), pointer conversions (7.11), function pointer conversions (7.13), and qualification conversions (7.5) are performed on both operands to bring them to their composite pointer type (8.2.2). If at least one of the operands is of pointer-to-member type, pointer-to-member conversions (7.12) and qualification conversions (7.5) are performed on both operands to bring them to their composite pointer type (8.2.2). If both operands are null pointer constants, but not both of integer type, the program is ill-formed.

7 If the composite pointer type is a function pointer type, a pointer-to-member type, or \( \text{std::nullptr\_t} \), the result is of type \( \text{std::strong\_equality} \); the result is \( \text{std::strong\_equality::equal} \) if the (possibly converted) operands compare equal (8.5.10) and \( \text{std::strong\_equality::unequal} \) if they compare unequal, otherwise the result of the operator is unspecified.

8 If the composite pointer type is an object pointer type, \( p <=> q \) is of type \( \text{std::strong\_ordering} \). If two pointer operands \( p \) and \( q \) compare equal (8.5.10), \( p <=> q \) yields \( \text{std::strong\_ordering::equal} \) if \( p \) and \( q \) compare unequal, \( p <=> q \) yields \( \text{std::strong\_ordering::less} \) if \( q \) compares greater than \( p \) and \( \text{std::strong\_ordering::greater} \) if \( p \) compares greater than \( q \) (8.5.9). Otherwise, the result is unspecified.

9 Otherwise, the program is ill-formed.

10 The five comparison category types (21.10.2) (the types \( \text{std::strong\_ordering} \), \( \text{std::strong\_equality} \), \( \text{std::weak\_ordering} \), \( \text{std::weak\_equality} \), and \( \text{std::partial\_ordering} \)) are not predefined; if the header \#include \(<\text{compare}>\) 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 (10.1.7.4) in a defaulted three-way comparison (15.9.2) or use of the built-in operator) – the program is ill-formed.

8.5.9 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

§ 8.5.9 120
The operands shall have arithmetic, enumeration, or pointer type. The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield false or true. The type of the result is bool.

The usual arithmetic conversions (8.3) are performed on operands of arithmetic or enumeration type. If both operands are pointers, pointer conversions (7.11) and qualification conversions (7.5) are performed to bring them to their composite pointer type (8.2). After conversions, the operands shall have the same type.

Comparing unequal pointers to objects is defined as follows:

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 compares greater.
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 compares greater provided the two members have the same access control (Clause 14) and provided their class is not a union.
3. Otherwise, neither pointer compares greater than the other.

If two operands p and q compare equal (8.5.10), p<q and p=q both yield true and p<=q both yield false. Otherwise, if a pointer p compares greater than a pointer q, p>=q and p>q both 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.

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.

8.5.10 Equality operators [expr.eq]

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 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.

2. If at least one of the operands is a pointer, pointer conversions (7.11), function pointer conversions (7.13), and qualification conversions (7.5) are performed on both operands to bring them to their composite pointer type (8.2). Comparing pointers is defined as follows:

   1. If one pointer represents the address of a complete object, and another pointer represents the address one past the last element of a different complete object, the result of the comparison is unspecified.
   2. Otherwise, if the pointers are both null, both point to the same function, or both represent the same address (6.7.2), they compare equal.
   3. Otherwise, the pointers compare unequal.

3. If at least one of the operands is a pointer to member, pointer-to-member conversions (7.12) and qualification conversions (7.5) are performed on both operands to bring them to their composite pointer type (8.2). Comparing pointers to members is defined as follows:

   1. If two pointers to members are both the null member pointer value, they compare equal.
   2. If only one of two pointers to members is the null member pointer value, they compare unequal.
   3. If either is a pointer to a virtual member function, the result is unspecified.
   4. If one refers to a member of class 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; };
```]

87) An object that is not an array element is considered to belong to a single-element array for this purpose; see 8.5.2.1. 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.

88) An object that is not an array element is considered to belong to a single-element array for this purpose; see 8.5.2.1.
int A::*bx = (int(A::*))&B::x;
int A::*cx = (int(A::*))&C::x;

bool b1 = (bx == cx); // unspecified

— end example

(3.5) — If both refer to (possibly different) members of the same union (12.3), they compare equal.

(3.6) — 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:

```cpp
struct B {
  int f();
};
struct L : B {}
struct R : B {}
struct D : L, R {}

int (B::*pb)() = &B::f;
int (L::*pl)() = pb;
int (R::*pr)() = pb;
int (D::*pdl)() = pl;
int (D::*pdr)() = pr;
bool x = (pdl == pdr); // false
bool y = (pb == pl); // true

— end example
```

4 Two operands of type std::nullptr_t or one operand of type std::nullptr_t and the other a null pointer constant compare equal.

5 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.

6 If both operands are of arithmetic or enumeration type, the usual arithmetic conversions (8.3) are performed
on both operands; each of the operators shall yield true if the specified relationship is true and false if it is
false.

### 8.5.11 Bitwise AND operator

```cpp
and-expression:
  equality-expression
  and-expression & equality-expression
```

1 The usual arithmetic conversions (8.3) are performed; the result is the bitwise AND function of the operands.
The operator applies only to integral or unscoped enumeration operands.

### 8.5.12 Bitwise exclusive OR operator

```cpp
exclusive-or-expression
  and-expression
  exclusive-or-expression ^ and-expression
```

1 The usual arithmetic conversions (8.3) are performed; the result is the bitwise exclusive OR function of the
operands. The operator applies only to integral or unscoped enumeration operands.

### 8.5.13 Bitwise inclusive OR operator

```cpp
inclusive-or-expression
  exclusive-or-expression
  inclusive-or-expression | exclusive-or-expression
```

1 The usual arithmetic conversions (8.3) are performed; the result is the bitwise inclusive OR function of its
operands. The operator applies only to integral or unscoped enumeration operands.
8.5.14 Logical AND operator

logical-and-expression:
  inclusive-or-expression
logical-and-expression && inclusive-or-expression

1 The `&&` operator groups left-to-right. The operands are both contextually converted to `bool` (Clause 7). The result is `true` if both operands are `true` and `false` otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is `false`.

2 The result is a `bool`. If the second expression is evaluated, 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.

8.5.15 Logical OR operator

logical-or-expression:
  logical-and-expression
logical-or-expression || logical-and-expression

1 The `||` operator groups left-to-right. The operands are both contextually converted to `bool` (Clause 7). 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.

8.5.16 Conditional operator

conditional-expression:
  logical-or-expression
conditional-expression ? expression : assignment-expression

1 Conditional expressions group right-to-left. The first expression is contextually converted to `bool` (Clause 7). 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` (8.5.17); 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.

[Note: This includes the case where both operands are `throw-expressions`. — end note]

(2.2) — Both the second and the third operands have type `void`; the result is of type `void` and is a prvalue.

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 (16.3.3.1) 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 (11.6.3) to an lvalue.

(4.2) — If `E2` is an xvalue, 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:
— if \( T_1 \) and \( T_2 \) are the same class type (ignoring cv-qualification), or one is a base class of the other, and \( T_2 \) is at least as cv-qualified as \( T_1 \), the target type is \( T_2 \),

— otherwise, the target type is the type that \( E_2 \) would have after applying the lvalue-to-rvalue (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) standard conversions.

Using this process, it is determined whether an implicit conversion sequence can be formed from the second operand to the target type determined for the third operand, and vice versa. If both sequences can be formed, or one can be formed but it is the ambiguous conversion sequence, the program is ill-formed. If no conversion sequence can be formed, the operands are left unchanged and further checking is performed as described below. Otherwise, if exactly one conversion sequence can be formed, that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this subclause. [ Note: The conversion might be ill-formed even if an implicit conversion sequence could be formed. — end note ]

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 (16.3.1.2, 16.6). 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 (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

— The second and third operands have the same type; the result is of that type and the result object is initialized using the selected operand.

— The second and third operands have arithmetic or enumeration type; the usual arithmetic conversions (8.3) are performed to bring them to a common type, and the result is of that type.

— One or both of the second and third operands have pointer type; pointer conversions (7.11), function pointer conversions (7.13), and qualification conversions (7.5) are performed to bring them to their composite pointer type (8.2). The result is of the composite pointer type.

— One or both of the second and third operands have pointer-to-member type; pointer to member conversions (7.12) and qualification conversions (7.5) are performed to bring them to their composite pointer type (8.2). The result is of the composite pointer type.

— Both the second and third operands have type \( \text{std::nullptr_t} \) or one has that type and the other is a null pointer constant. The result is of type \( \text{std::nullptr_t} \).

8.5.17 Throwing an exception [expr.throw]

A throw-expression is of type \( \text{void} \).

Evaluating a throw-expression with an operand throws an exception (18.1); 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 \( T \)”.

A throw-expression with no operand rethrows the currently handled exception (18.3). 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:

```c++
try {
  // ...
} catch (...) {
  // catch all exceptions
  // respond (partially) to exception
  throw;  // pass the exception to some other handler
}
```

— end example ]
4 If no exception is presently being handled, evaluating a throw-expression with no operand calls std::terminate() (18.5.1).

8.5.18 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]

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression

assignment-operator: one of
  = *= /= %= += -= >>= <<= &= ^= |=

2 In simple assignment (=), the value of the expression replaces that of the object referred to by the left operand.

3 If the left operand is not of class type, the expression is implicitly converted (Clause 7) 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 (15.8, 16.5.3).

5 [ Note: For class objects, assignment is not in general the same as initialization (11.6, 15.1, 15.6, 15.8). — 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 E1 op = E2 is equivalent to E1 = E1 op E2 except that E1 is evaluated only once. In += and -=, E1 shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined object type. In all other cases, E1 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 8.2.1. — 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 (16.5.3, 16.3).

[ 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]

8.5.19 Comma operator

The comma operator groups left-to-right.
A pair of expressions separated by a comma is evaluated left-to-right; the left expression is a discarded-value expression (8.2). 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 (15.2), the result is a temporary expression.

In contexts where comma is given a special meaning, [Example: in lists of arguments to functions (8.5.1.2) and lists of initializers (11.6) — 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]

### 8.6 Constant expressions

Certain contexts require expressions that satisfy additional requirements as detailed in this subclause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements, assuming that copy elision is performed, are called constant expressions. [Note: Constant expressions can be evaluated during translation. — end note]

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:

1. **this** (8.4.2), except in a constexpr function or a constexpr constructor that is being evaluated as part of \( e \);
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 (15.4) [Note: Overload resolution (16.3) is applied as usual — end note];
3. an invocation of an undefined constexpr function or an undefined constexpr constructor;
4. an invocation of an instantiated constexpr function or constexpr constructor that fails to satisfy the requirements for a constexpr function or constexpr constructor (10.1.5);
5. an expression that would exceed the implementation-defined limits (see Annex B);
6. an operation that would have undefined behavior as specified in Clause 4 through Clause 19 of this document [Note: including, for example, signed integer overflow (8.2), certain pointer arithmetic (8.5.6), division by zero (8.5.5), or certain shift operations (8.5.7) — end note];
7. an lvalue-to-rvalue conversion (7.1) unless it is applied to
   7.1 a non-volatile glvalue of integral or enumeration type that refers to a complete non-volatile const object with a preceding initialization, initialized with a constant expression, or
   7.2 a non-volatile glvalue that refers to a subobject of a string literal (5.13.5), or
   7.3 a non-volatile glvalue that refers to a non-volatile object defined with constexpr, or that refers to a non-mutable subobject of such an object, or
   7.4 a non-volatile glvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of \( e \);
8. an lvalue-to-rvalue conversion (7.1) that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;
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 \( e \);
10. an assignment expression (8.5.18) or invocation of an assignment operator (15.8) that would change the active member of a union;
— an id-expression that refers to a variable or data member of reference type unless the reference has a preceding initialization and either

— it is initialized with a constant expression or

— its lifetime began within the evaluation of e;

— in a lambda-expression, a reference to this or to a variable with automatic storage duration defined outside that lambda-expression, where the reference would be an odr-use (6.2, 8.4.5); [Example:

```cpp
void g() {
    const int n = 0;
    [=] {
        constexpr int i = n; // OK, n is not odr-used and not captured here
        constexpr int j = *n; // ill-formed, &n would be an odr-use of n
    };
}

— end example] [ Note: If the odr-use occurs in an invocation of a function call operator of a closure type, it no longer refers to this or to an enclosing automatic variable due to the transformation (8.4.5.2) of the id-expression into an access of the corresponding data member. [ Example:

```cpp
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.
static_assert(bind(monad(2))(monad)() == monad(2)());

— end example] — end note]}

— a conversion from type cv void* to a pointer-to-object type;

— a dynamic cast (8.5.1.7);

— a reinterpret_cast (8.5.1.10);

— a pseudo-destructor call (8.5.1.4);

— modification of an object (8.5.18, 8.5.1.6, 8.5.2.2) unless it is applied to a non-volatile lvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of e;

— a typeid expression (8.5.1.8) whose operand is a glvalue of a polymorphic class type;

— a new-expression (8.5.2.4);

— a delete-expression (8.5.2.5);

— a three-way comparison (8.5.8) comparing pointers that do not point to the same complete object or to any subobject thereof;

— a relational (8.5.9) or equality (8.5.10) operator where the result is unspecified; or

— a throw-expression (8.5.17).

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 20 through Clause 33 of this document, it is unspecified whether e is a core constant expression.

[ Example:

```cpp
int x; // not constant
struct A {
    constexpr A(bool b) : m(b?42:x) { }
    int m;
};
constexpr int v = A(true).m; // OK: constructor call initializes m with the value 42
constexpr int w = A(false).m; // error: initializer for m is x, which is non-constant
constexpr int f1(int k) {
    constexpr int x = k; // error: x is not initialized by a constant expression
```
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)

— end example ]

3 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 (12.2.4), as enumerator initializers if the underlying type is not fixed (10.2), and as alignments (10.6.2). — end note ]

4 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 (Clause 7) 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);
aligas(a) int n;  // error: ambiguous conversion
struct B { int n : a; };  // error: ambiguous conversion
— end example ]

5 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

(5.1) user-defined conversions,
(5.2) lvalue-to-rvalue conversions (7.1),
(5.3) array-to-pointer conversions (7.2),
(5.4) function-to-pointer conversions (7.3),
(5.5) qualification conversions (7.5),
(5.6) integral promotions (7.6),
(5.7) integral conversions (7.8) other than narrowing conversions (11.6.4),
(5.8) null pointer conversions (7.11) from std::nullptr_t,
(5.9) null member pointer conversions (7.12) from std::nullptr_t, and
(5.10) function pointer conversions (7.13),

§ 8.6 128
and where the reference binding (if any) binds directly. \[ Note: \] Such expressions may be used in new expressions (8.5.2.4), as case expressions (9.4.2), as enumerator initializers if the underlying type is fixed (10.2), as array bounds (11.3.4), and as non-type template arguments (17.3). \[ — end note \] A contextually converted constant expression of type bool is an expression, contextually converted to bool (Clause 7), where the converted expression is a constant expression and the conversion sequence contains only the conversions above.

6 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:

(6.1) — if the value is an object of class type, each non-static data member of reference type refers to an entity that is a permitted result of a constant expression,

(6.2) — if the value is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object (8.5.6), the address of a function, or a null pointer value, and

(6.3) — 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 is either not a temporary object or is a temporary object whose value satisfies the above constraints, or it is a function.

7 \[ Note: \] 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.\[89\] \[ Example:\]

```c
bool f() {
    char array[1 + int(1 + 0.2 - 0.1 - 0.1)]; // Must be evaluated during translation
    int size = 1 + int(1 + 0.2 - 0.1 - 0.1); // May be evaluated at runtime
    return sizeof(array) == size;
}
```

It is unspecified whether the value of f() will be true or false. \[ — end example \] \[ — end note \]

8 An expression is potentially constant evaluated if it is:

(8.1) — a potentially-evaluated expression (6.2),

(8.2) — a constraint-expression, including one formed from the constraint-logical-or-expression of a requires-clause,

(8.3) — an immediate subexpression of a braced-init-list,\[90\]

(8.4) — an expression of the form & cast-expression that occurs within a templated entity,\[91\] or

(8.5) — a subexpression of one of the above that is not a subexpression of a nested unevaluated operand.

A function or variable is needed for constant evaluation if it is:

(8.6) — a constexpr function that is named by an expression (6.2) that is potentially constant evaluated, or

(8.7) — 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.

\[89\] Nonetheless, implementations should provide consistent results, irrespective of whether the evaluation was performed during translation and/or during program execution.

\[90\] Constant evaluation may be necessary to determine whether a narrowing conversion is performed (11.6.4).

\[91\] Constant evaluation may be necessary to determine whether such an expression is value-dependent (17.7.2.3).
9 Statements

1 Except as indicated, statements are executed in sequence.

```plaintext
statement:          labeled-statement
                  attribute-specifier-seqopt expression-statement
                  attribute-specifier-seqopt compound-statement
                  attribute-specifier-seqopt selection-statement
                  attribute-specifier-seqopt iteration-statement
                  attribute-specifier-seqopt jump-statement
                  declaration-statement
                  attribute-specifier-seqopt try-block

init-statement:     expression-statement
                  simple-declaration

condition:          expression
                  attribute-specifier-seqopt decl-specifier-seq declarator brace-or-equal-initializer
```

The optional attribute-specifier-seq appertains to the respective statement.

2 The rules for conditions apply both to selection-statements and to the for and while statements (9.5). 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 10.1.7.4.

3 A name introduced by a declaration in a condition (either introduced by 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:

```plaintext
if (int x = f()) {
  int x;  // ill-formed, redeclaration of x
} else {
  int x;  // ill-formed, redeclaration of x
}
—end example]
```

4 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 (Clause 7). If that conversion is ill-formed, the program is ill-formed. The value of a condition that is an initialized declaration in a switch statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted to integral or enumeration type otherwise. The value of a condition that is an expression is the value of the expression, contextually converted to bool for statements other than switch; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply “the condition” where the usage is unambiguous.

5 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.

6 In the decl-specifier-seq of a condition, each decl-specifier shall be either a type-specifier or constexpr.

9.1 Labeled statement

1 A statement can be labeled.

```plaintext
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.

9.2 Expression statement

Expression statements have the form

expression-statement:
  expression-opt ;

The expression is a discarded-value expression (8.2). 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 (9.5.1). — end note]

9.3 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-seq-opt }

statement-seq:
  statement
  statement-seq statement

A compound statement defines a block scope (6.3). [Note: A declaration is a statement (9.7). — end note]

9.4 Selection statements

Selection statements choose one of several flows of control.

selection-statement:
  if constexpr opt ( init-statement-opt condition ) statement
  if constexpr opt ( init-statement-opt condition ) statement else statement
  switch ( init-statement-opt condition ) statement

See 11.3 for the optional attribute-specifier-seq in a condition. [Note: An init-statement ends with a semicolon. — end note] In Clause 9, the term substatement refers to the contained statement or statements that appear in the syntax notation. 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:

```c
if (x)
  int i;
```

can be equivalently rewritten as

```c
if (x) {
  int i;
}
```

Thus after the if statement, i is no longer in scope. — end example]

9.4.1 The if statement

If the condition (9.4) 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.\(^{92}\)

\(^{92}\) In other words, the else is associated with the nearest un-elsed if.
2 If the if statement is of the form `if constexpr`, the value of the condition shall be a contextually converted constant expression of type `bool` (8.6); 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 17), 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 (9.4.2) within the same if statement. A label (9.1) declared in a substatement of a constexpr if statement shall only be referred to by a statement (9.6.4) 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]
```

3 An if statement of the form

```cpp
    if constexpr_opt ( init-statement condition ) statement
```

is equivalent to

```cpp
    {
        init-statement
        if constexpr_opt ( condition ) statement
    }
```

and an if statement of the form

```cpp
    if constexpr_opt ( init-statement condition ) statement else statement
```

is equivalent to

```cpp
    {
        init-statement
        if constexpr_opt ( condition ) statement else statement
    }
```

except that names declared in the `init-statement` are in the same declarative region as those declared in the `condition`.

9.4.2 The switch statement [stmt.switch]

1 The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (Clause 7) to an integral or enumeration type. If the (possibly converted) type is subject to integral promotions (7.6), 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:

```cpp
    case constant-expression :
```

where the `constant-expression` shall be a converted constant expression (8.6) 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.

Switch statements can be nested; a case or default label is associated with the smallest switch enclosing it.

When the switch statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of 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.

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. 9.6.1. [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]

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.

9.5 Iteration statements

Iteration statements specify looping.

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

See 11.3 for the optional attribute-specifier-seq in a for-range-declaration. [Note: An init-statement ends with a semicolon. — end note]

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:

while (--x >= 0)
  int i;

can be equivalently rewritten as

while (--x >= 0) {
  int i;
}

Thus after the while statement, i is no longer in scope. — end example]

If a name introduced in an init-statement or for-range-declaration is redeclared in the outermost block of the substatement, the program is ill-formed. [Example:

void f() {
  for (int i = 0; i < 10; ++i)
    int i = 0; // error: redeclaration

§ 9.5
for (int i : {1, 2, 3})
    int i = 1;  // error: redeclaration

—end example]}

9.5.1 The while statement [stmt.while]

1 In the while statement the substatement is executed repeatedly until the value of the condition (9.4) 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 of the form

   while (T t = x) statement

is equivalent to

   label:
   {
     // start of condition scope
     T t = x;
     if (t) {
       statement
       goto label;
     }
   }
   // end of condition scope

The variable created in a condition is destroyed and created with each iteration of the loop. [Example:

   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]

9.5.2 The do statement [stmt.do]

1 The expression is contextually converted to bool (Clause 7); 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.

9.5.3 The for statement [stmt.for]

1 The for statement

   for (init-statement condition_opt ; expression_opt ) statement

is equivalent to

   {
     init-statement
     while (condition) {
       statement
       expression;
     }
   }

except that names declared in the init-statement are in the same declarative region as those declared in the condition, and except that a continue in statement (not enclosed in another iteration statement) will execute expression before re-evaluating condition. [Note: Thus the first statement specifies initialization for the loop; the condition (9.4) 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]

9.5.4 The range-based for statement
[stmt.ranged]

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 + __bound, respectively, where __bound 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 either (or both) finds 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:

```c
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
    x *= 2;
—end example]
```

2 In the decl-specifier-seq of a for-range-declaration, each decl-specifier shall be either a type-specifier or constexpr. The decl-specifier-seq shall not define a class or enumeration.

9.6 Jump statements
[stmt.jump]

Jump statements unconditionally transfer control.
jump-statement:
  break ;
  continue ;
  return expr-or-braced-init-listopt ;
  goto identifier ;

2 On exit from a scope (however accomplished), objects with automatic storage duration (6.6.4.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [Note: For temporaries, see 15.2. — 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 9.7 for transfers into blocks). [Note: However, the program can be terminated (by calling std::exit() or std::abort() (21.5), for example) without destroying class objects with automatic storage duration. — end note]

9.6.1 The break statement

1 The break statement shall occur only in an iteration-statement or a switch statement and causes termination of the smallest enclosing iteration-statement or switch statement; control passes to the statement following the terminated statement, if any.

9.6.2 The continue statement

1 The continue statement shall occur only in an iteration-statement and causes control to pass to the loop-continuation portion of the smallest enclosing iteration-statement, that is, to the end of the loop. More precisely, in each of the statements

```cpp
while (foo) {
  do {
    // ...
    // ...
  }
  while (foo);
}
```

a continue not contained in an enclosed iteration statement is equivalent to goto contin.

9.6.3 The return statement

1 A function returns to its caller by the return statement.

2 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 (15.1), or a destructor (15.4). A return statement with an operand of type void shall be used only in a function whose return type is cv void. A return statement with any other operand shall be used only in a function whose return type is not cv void; the return statement initializes the glvalue result or prvalue result object of the (explicit or implicit) function call by copy-initialization (11.6) 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 (15.8). — end note] [Example:

```cpp
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

— end example] Flowing off the end of a constructor, a destructor, or a function with a cv void return type is equivalent to a return with no operand. Otherwise, flowing off the end of a function other than main (6.8.3.1) 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 return statement, which, in turn, is sequenced before the destruction of local variables (9.6) of the block enclosing the return statement.

9.6.4 The goto statement

1 The goto statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label (9.1) located in the current function.
9.7 Declaration statement

A declaration statement introduces one or more new identifiers into a block; it has the form

```
decoration-statement:
  block-declaration
```

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

Variables with automatic storage duration (6.6.4.3) are initialized each time their declaration-statement is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (9.6).

It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps from a point where a variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has scalar type, class type with a trivial default constructor and a trivial destructor, a cv-qualified version of one of these types, or an array of one of the preceding types and is declared without an initializer (11.6). [Example:

```c
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 example]

Dynamic initialization of a block-scope variable with static storage duration (6.6.4.1) or thread storage duration (6.6.4.2) 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]

The destructor for a block-scope object with static or thread storage duration will be executed 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]

9.8 Ambiguity resolution

There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (8.5.1.3) 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 (10.1.7),

```c
T(a)->m = 7;   // expression-statement
T(a)++;       // expression-statement
T(a,5)<<c;    // expression-statement
```

93) The transfer from the condition of a switch statement to a case label is considered a jump in this respect.

94) 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.

§ 9.8
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.  

```cpp
class T {
    // ...
    public:
    T();
    T(int);
    T(int, int);
};
T(a);
// declaration
T(*b()); // declaration
T(c)=7;  // declaration
T(d), e, f=3; // declaration
extern int h;
T(g)(h,2);  // declaration
— end example] — end note]
```

The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are type-names or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a type-name. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [Note: This can occur only when the name is declared earlier in the declaration. — end note] [Example:

```cpp
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));  // T2 will be declared as a variable of type T1, but this will not
// allow the last part of the declaration to parse properly,
// since it depends on T2 being a type-name
}
— end example]
```
10 Declarations [dcl.dcl]

1 Declarations generally specify how names are to be interpreted. Declarations have the form

```
declaration-seq:
declaration
  declaration-seq declaration

declaration:
  block-declaration
  nodeclspec-function-declaration
  function-definition
  template-declaration
  deduction-guide
  explicit-instantiation
  explicit-specialization
  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-seq_opt declarator ;

alias-declaration:
  using identifier attribute-specifier-seq_opt = defining-type-id ;

simple-declaration:
  decl-specifier-seq init-declarator-list_opt ;
  attribute-specifier-seq decl-specifier-seq init-declarator-list ;
  attribute-specifier-seq_opt decl-specifier-seq ref-qualifier_opt [ identifier-list ] initializer ;

static_assert-declaration:
  static_assert ( constant-expression ) ;
  static_assert ( constant-expression , string-literal ) ;

empty-declaration:
  ;

attribute-declaration:
  attribute-specifier-seq ;
```

[Note: asm-fontions are described in 10.4, and linkage-specifications are described in 10.5. Function-declarations are described in 11.4 and template-declarations and deduction-guides are described in Clause 17. Namespace-definitions are described in 10.3.1, using-declarations are described in 10.3.3 and using-directives are described in 10.3.4. — end note]

2 A simple-declaration or nodeclspec-function-declaration of the form

```
attribute-specifier-seq_opt decl-specifier-seq_opt init-declarator-list_opt ;
```

is divided into three parts. Attributes are described in 10.6. decl-specifiers, the principal components of a decl-specifier-seq, are described in 10.1. declarators, the components of an init-declarator-list, are described in Clause 11. The attribute-specifier-seq appertains to each of the entities declared by the declarators of the init-declarator-list. [Note: In the declaration for an entity, attributes appertaining to that entity may appear at the start of the declaration and after the declarator-id for that declaration. — end note]

Example:
[noreturn] void f [noreturn] ();    // OK
—end example]

3 Except where otherwise specified, the meaning of an attribute-declaration is implementation-defined.

4 A declaration occurs in a scope (6.3); 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 10 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are not nested within scopes nested within the declaration.

5 In a simple-declaration, the optional init-declarator-list can be omitted only when declaring a class (Clause 12) or enumeration (10.2), that is, when the decl-specifier-seq contains either a class-specifier, an elaborated-typeSpecifier with a class-key (12.1), 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:
enum { };    // ill-formed
typedef class { }; // ill-formed
—end example]

6 In a static_assert-declaration, the constant-expression shall be a contextually converted constant expression of type bool (8.6). 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:
static_assert(char(-1) < 0, "this library requires plain 'char' to be signed");
—end example]

7 An empty-declaration has no effect.

8 A simple-declaration with an identifier-list is called a structured binding declaration (11.5). The decl-specifier-seq shall contain only the typeSpecifier auto (10.1.7.4) and cv-qualifiers. The initializer shall be of the form "= assignment-expression", of the form "{ assignment-expression }", or of the form "( assignment-expression )", where the assignment-expression is of array or non-union class type.

9 Each init-declarator in the init-declarator-list contains exactly one declarator-id, which is the name declared by that init-declarator and hence one of the names declared by the declaration. The defining-type-specifiers (10.1.7) in the decl-specifier-seq and the recursive declarator structure of the init-declarator describe a type (11.3), which is then associated with the name being declared by the init-declarator.

10 If the decl-specifier-seq contains the typedef specifier, the declaration is called a typedef declaration and the name of each init-declarator is declared to be a typedef-name, synonymous with its associated type (10.1.3). 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 (11.3.5) 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 (11.6) to be done.

12 A nodclspec-function-declaration shall declare a constructor, destructor, or conversion function.95 [Note: A nodclspec-function-declaration can only be used in a template-declaration (Clause 17), explicit-instantiation (17.8.2), or explicit-specialization (17.8.3). —end note]
The optional attribute-specifier-seq in a decl-specifier-seq appertains to the type determined by the preceding decl-specifiers (11.3). 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.

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:

```c
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,

```c
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:

```c
void h(unsigned Pc);
// void h(unsigned int)
void k(unsigned int Pc);
// void k(unsigned int)
```

— end example] — end note]

### 10.1.1 Storage class specifiers

The storage class specifiers are

```c
storage-class-specifier:
static
thread_local
extern
mutable
```

At most one storage-class-specifier shall appear in a given decl-specifier-seq, except that thread_local may appear with static or extern. If thread_local appears in any declaration of a variable it shall be present in all declarations of that entity. If a storage-class-specifier appears in a decl-specifier-seq, there can be no typedef specifier in the same decl-specifier-seq and the init-declarator-list or member-declarator-list of the declaration shall not be empty (except for an anonymous union declared in a named namespace or in the global namespace, which shall be declared static (12.3.1)). 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 (17.8.3) or an explicit instantiation (17.8.2) 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 (6.6.4.3). — end note]

3 The thread_local specifier indicates that the named entity has thread storage duration (6.6.4.2). It shall be applied only to the names of variables of namespace or block scope and to the names of static data members.
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 can be applied only to names of variables and functions and to anonymous unions (12.3.1). 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 (6.6.4.1), unless accompanied by the `thread_local` specifier, which declares the variable to have thread storage duration (6.6.4.2). A `static` specifier can be used in declarations of class members; 12.2.3 describes its effect. For the linkage of a name declared with a `static` specifier, see 6.5.

5 The `extern` specifier can be applied only to the names of variables and functions. The `extern` specifier cannot be used in the declaration of class members or function parameters. 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();
inline void h();     // external linkage

inline void l();     // external linkage
void l();

inline void m();     // external linkage
extern void m();

static void n();     // internal linkage
inline void n();     // internal linkage

static int a;        // a has internal linkage
int a;               // error: two definitions

static int b;        // b has internal linkage
extern int b;        // b still has internal linkage

int c;               // c has external linkage
static int c;        // error: inconsistent linkage

extern int d;        // d has external linkage
static int d;        // error: inconsistent linkage
```
—end example]

7 The name of a declared but undefined class can be used in an `extern` declaration. Such a declaration can only be used in ways that do not require a complete class type. [Example:

```c
struct S;
extern S a;
extern S f();
extern void g(S);

void h() {
    g(a);     // error: S is incomplete
    f();      // error: S is incomplete
}
```
The `mutable` specifier shall appear only in the declaration of a non-static data member (12.2) whose type is neither const-qualified nor a reference type.  

```cpp
class X {
    mutable const int* p; // OK
    mutable int* const q; // ill-formed
};
```

The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is `const` (10.1.7.1).

### 10.1.2 Function specifiers

Function specifiers can be used only in function declarations.

```cpp
function-specifier:
    virtual
    explicit
```

1. The `virtual` specifier shall be used only in the initial declaration of a non-static class member function; see 13.3.
2. The `explicit` specifier shall be used only in the declaration of a constructor or conversion function within its class definition; see 15.3.1 and 15.3.2.

### 10.1.3 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` (11.3.5) nor in the `decl-specifier-seq` of a `function-definition` (11.4). If a `typedef` specifier appears in a declaration without a `declarator`, the program is ill-formed.

```cpp
typedef-name:
    identifier
```

A name declared with the `typedef` specifier becomes a `typedef-name`. Within the scope of its declaration, a `typedef-name` is syntactically equivalent to a keyword and names the type associated with the identifier in the way described in Clause 11. A `typedef-name` is thus a synonym for another type. A `typedef-name` does not introduce a new type the way a class declaration (12.1) or enum declaration does.  

```cpp
typedef int MILES, *KLICKSP;

MILES distance;
extern KLICKSP metricp;
```

The constructions are all correct declarations; the type of `distance` is `int` and that of `metricp` is “pointer to int”.  

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.  

```cpp
using handler_t = void (*)(int);
using handler_t ignore;
extern void (*ignore)(int); // redeclare ignore
using cell = pair<void*, cell*>; // ill-formed
```

In a given non-class scope, a `typedef` specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers.  

```cpp
typedef struct s { /* ... */ } s;
typedef int I;
```
typedef int I;
typeid I I;
—end example]

4 In a given class scope, a typedef specifier can be used to redefine any class-name declared in that scope that is not also a typedef-name to refer to the type to which it already refers. [Example:

```c
struct S {
    typedef struct A { } A;     // OK
    typedef struct B B;         // OK
    typedef A A;                // error
};
—end example]
```

5 If a typedef specifier is used to redefine in a given scope an entity that can be referenced using an elaborated-type-specifier, the entity can continue to be referenced by an elaborated-type-specifier or as an enumeration or class name in an enumeration or class definition respectively. [Example:

```c
struct S;
typedef struct S S;         // OK
int main() {
    struct S* p;              // OK
    struct S { };             // OK
—end example]
```

6 In a given scope, a typedef specifier shall not be used to redefine the name of any type declared in that scope to refer to a different type. [Example:

```c
class complex { /* ... */ };  // error: redefinition
typedef int complex;         // error: redefinition
—end example]
```

7 Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a typedef-name that is declared in that scope and refers to a type other than the class or enumeration itself. [Example:

```c
typedef int complex;
class complex { /* ... */ };  // error: redefinition
—end example]
```

8 [Note: A typedef-name that names a class type, or a cv-qualified version thereof, is also a class-name (12.1). If a typedef-name is used to identify the subject of an elaborated-type-specifier (10.1.7.3), a class definition (Clause 12), a constructor declaration (15.1), or a destructor declaration (15.4), the program is ill-formed. —end note] [Example:

```c
struct S {
    S();
    ~S();
};
typedef struct S T;
S a = T();          // OK
struct T * p;       // error
—end example]
```

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:

```c
typedef struct {} *ps, S;   // S is the class name for linkage purposes
typedef decltype([]() {}) C; // the closure type has no name for linkage purposes
—end example]
```
10.1.4 The friend specifier  

The friend specifier is used to specify access to class members; see 14.3.

10.1.5 The constexpr specifier  

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. A function or static data member declared with the constexpr specifier is implicitly an inline function or variable (10.1.6). If any declaration of a function or function template has a constexpr specifier, then all its declarations shall contain the constexpr specifier. [Note: An explicit specialization can differ from the template declaration with respect to the constexpr specifier. —end note] [Note: Function parameters cannot be declared constexpr. —end note] [Example:

```cpp
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
};
cssexpr pixelpixel::pixel(int a)
  : x(a), y(x) // OK: definition
  { square(x); }
cssexpr pixel small(2); // error: square not defined, so small(2) // not constant (8.6) so constexpr not satisfied
```

A constexpr specifier used in the declaration of a function that is not a constructor declares that function to be a constexpr function. Similarly, a constexpr specifier used in a constructor declaration declares that constructor to be a constexpr constructor.

The definition of a constexpr function shall satisfy the following requirements:

1. it shall not be virtual (13.3);
2. its return type shall be a literal type;
3. each of its parameter types shall be a literal type;
4. its function-body shall be = delete, = default, or a compound-statement that does not contain
   1. an asm-definition,
   2. a goto statement,
   3. an identifier label (9.1),
   4. a try-block, or
   5. a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed.

[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;
```
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;

4.3 its function-body shall not be a function-try-block.

In addition, either its function-body shall be = delete, or it shall satisfy the following requirements:

4.4 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;

4.5 every non-variant non-static data member and base class subobject shall be initialized (15.6.2);

4.6 if the class is a union having variant members (12.3), exactly one of them shall be initialized;

4.7 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;

4.8 for a non-delegating constructor, every constructor selected to initialize non-static data members and base class subobjects shall be a constexpr constructor;

4.9 for a delegating constructor, the target constructor shall be a constexpr constructor.

[Example:
  struct Length {
    constexpr explicit Length(int i = 0) : val(i) { }
  private:
    int val;
  };
  — end example]

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 (8.6), 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
  // value-to-value conversion on non-constant global
};  // end example

6 If the instantiated template specialization of a constexpr function template or member function of a class
template would fail to satisfy the requirements for a constexpr function or constexpr constructor, that
specialization is still a constexpr function or constexpr constructor, even though a call to such a function
cannot appear in a constant expression. If no specialization of the template would satisfy the requirements
for a constexpr function or constexpr constructor when considered as a non-template function or constructor,
the template is ill-formed, no diagnostic required.

7 A call to a constexpr function produces the same result as a call to an equivalent non-constexpr function in
all respects except that

(7.1) — a call to a constexpr function can appear in a constant expression (8.6) and

(7.2) — copy elision is mandatory in a constant expression (15.8).

8 The constexpr specifier has no effect on the type of a constexpr function or a constexpr constructor.
[Example:
constexpr int bar(int x, int y)  // OK
{ return x + y + x*y; }
// ...
int bar(int x, int y)  // error: redefinition of bar
{ return x * 2 + 3 * y; }
—end example]

9 A constexpr specifier used in an object declaration declares the object as const. Such an object shall
have literal type and shall be initialized. In any constexpr variable declaration, the full-expression of the
initialization shall be a constant expression (8.6). [Example:
struct pixel {
  int x, y;
};
cconstexpr pixel ur = { 1294, 1024 };  // OK
cconstexpr pixel origin;  // error: initializer missing
—end example]

10.1.6 The inline specifier  [dcl.inline]

1 The inline specifier can be applied only to the declaration or definition of a variable or function.

2 A function declaration (11.3.5, 12.2.1, 14.3) with an inline specifier declares an inline function. The inline
specifier indicates to the implementation that inline substitution of the function body at the point of call is
to be preferred to the usual function call mechanism. An implementation is not required to perform this
inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for
inline functions specified in this subclause shall still be respected.

3 A variable declaration with an inline specifier declares an inline variable.

4 A function defined within a class definition is an inline function.

5 The inline specifier shall not appear on a block scope declaration. If the inline specifier is used in a
friend function declaration, that declaration shall be a definition or the function shall have previously been
declared inline.

6 An inline function or variable shall be defined in every translation unit in which it is odr-used and shall
have exactly the same definition in every case (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 the
definition of a function or variable appears in a translation unit before its first declaration as inline, the
program is ill-formed. If a function or variable with external linkage is declared inline in one translation
unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required. An
inline function or variable with external linkage shall have the same address in all translation units. [Note:

96) The inline keyword has no effect on the linkage of a function.
A static local variable in an inline function with external linkage always refers to the same object. A type defined within the body of an inline function with external linkage is the same type in every translation unit. — end note]

10.1.7 Type specifiers [dcl.type]

1 The type-specifiers are

- type-specifier:
  - simple-type-specifier
  - elaborated-type-specifier
  - typename-specifier
  - cv-qualifier

- type-specifier-seq:
  - type-specifier attribute-specifier-seqopt
  - type-specifier type-specifier-seq

- defining-type-specifier:
  - type-specifier
  - class-specifier
  - enum-specifier

- defining-type-specifier-seq:
  - defining-type-specifier attribute-specifier-seqopt
  - defining-type-specifier defining-type-specifier-seq

The optional attribute-specifier-seq in a type-specifier-seq or a defining-type-specifier-seq appertains to the type denoted by the preceding type-specifiers or defining-type-specifiers (11.3). The attribute-specifier-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-type-specifier is allowed in the complete decl-specifier-seq of a declaration or in a defining-type-specifier-seq, and at most one type-specifier is allowed in a type-specifier-seq. The only exceptions to this rule are the following:

1. const can be combined with any type specifier except itself.
2. volatile can be combined with any type specifier except itself.
3. signed or unsigned can be combined with char, long, short, or int.
4. short or long can be combined with int.
5. long can be combined with double.

3 Except in a declaration of a constructor, destructor, or conversion function, at least one defining-type-specifier that is not a cv-qualifier shall appear in a complete type-specifier-seq or a complete decl-specifier-seq. For example, these could be introduced by typedefs. — end note]

10.1.7.1 The cv-qualifiers [dcl.type.cv]

1 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 11.3.5 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 (10.1.1) and its usability in constant expressions (8.6). As described in 11.6, 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 (8.5.1.11). — end note]

97 There is no special provision for a decl-specifier-seq that lacks a type-specifier or that has a type-specifier that only specifies cv-qualifiers. The “implicit int” rule of C is no longer supported.
Except that any class member declared `mutable` (10.1.1) can be modified, any attempt to modify a `const` object during its lifetime (6.6.3) results in undefined behavior. [Example:

```cpp
const int ci = 3; // cv-qualified (initialized as required)
ci = 4; // ill-formed: attempt to modify const
int i = 2;
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*
*i = 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,

```cpp
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
```

The semantics of an access through a `volatile` glvalue are implementation-defined. If an attempt is made to access an object defined with a `volatile`-qualified type through the use of a non-`volatile` glvalue, the behavior is undefined.

[Note: `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. Furthermore, for some implementations, `volatile` might indicate that special hardware instructions are required to access the object. See 6.8.1 for detailed semantics. In general, the semantics of `volatile` are intended to be the same in C++ as they are in C. —end note]

10.1.7.2 Simple type specifiers [dcl.type.simple]

The simple type specifiers are
simple-type-specifier:
  nested-name-specifier_opt type-name
  nested-name-specifier template simple-template-id
  nested-name-specifier_opt template-name
  char
  char16_t
  char32_t
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void
  auto
dcltype-specifier

type-name:
  class-name
  enum-name
  typedef-name
  simple-template-id
dcltype-specifier:
  dcltype ( expression )
dcltype ( auto )

2 The simple-type-specifier auto is a placeholder for a type to be deduced (10.1.7.4). A type-specifier of the form typename_opt nested-name-specifier_opt template-name is a placeholder for a deduced class type (10.1.7.5). The template-name shall name a class template that is not an injected-class-name. 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 11 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] For an expression e, the type denoted by dcltype(e) is defined as follows:

(4.1) — if e is an unparenthesized id-expression naming a structured binding (11.5), dcltype(e) is the referenced type as given in the specification of the structured binding declaration;

(4.2) — otherwise, if e is an unparenthesized id-expression or an unparenthesized class member access (8.5.1.5), dcltype(e) is the type of the entity named by e. If there is no such entity, or if e names a set of overloaded functions, the program is ill-formed;

(4.3) — otherwise, if e is an xvalue, dcltype(e) is T&&, where T is the type of e;

(4.4) — otherwise, if e is an lvalue, dcltype(e) is T&, where T is the type of e;

(4.5) — otherwise, dcltype(e) is the type of e.

The operand of the dcltype specifier is an unevaluated operand (8.2).

[Example:

```c
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
dcltype(foo()) x1 = 17; // type is const int&&
dcltype(i) x2; // type is int
dcltype(a->x) x3; // type is double
dcltype((a->x)) x4 = x3; // type is const double&
```]
Table 11 — `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 17.2</td>
</tr>
<tr>
<td><code>template-name</code></td>
<td>placeholder for a type to be deduced</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>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>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 int</code></td>
<td>“long 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</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>
<tr>
<td><code>auto</code></td>
<td>placeholder for a type to be deduced</td>
</tr>
<tr>
<td><code>decltype(auto)</code></td>
<td>placeholder for a type to be deduced</td>
</tr>
<tr>
<td><code>decltype(expression)</code></td>
<td>the type as defined below</td>
</tr>
</tbody>
</table>

— end example]  [Note: The rules for determining types involving `decltype(auto)` are specified in 10.1.7.4. — end note]

If the operand of a `decltype-specifier` is a prvalue, the temporary materialization conversion is not applied (7.4) and no result object is provided for the prvalue. The type of the prvalue may be incomplete. [Note: As a result, storage is not allocated for the prvalue and it is not destroyed. Thus, a class type is not instantiated as a result of being the type of a function call in this context. In this context, the common purpose of writing the expression is merely to refer to its type. In that sense, a `decltype-specifier` is analogous to a use of a `typedef-name`, so the usual reasons for requiring a complete type do not apply. In particular, it is not necessary to allocate storage for a temporary object or to enforce the semantic constraints associated with invoking the type’s destructor. — end note]  [Note: Unlike the preceding rule, parentheses have no special meaning in this context. — end note]  [Example:

```c
  template<class T> struct A { ~A() = delete; };
```
template<class T> auto h()
    -> A<T>;
template<class T> auto i(T)
    // identity
    -> T;
template<class T> auto f(T)
    // #1
    -> decltype(i(h<T>()));
    // forces completion of A<T> and implicitly uses A<T>::~A()
    // for the temporary introduced by the use of h().
    // (A temporary is not introduced as a result of the use of i().)
    // for the temporary introduced by the use of f().
    // (A temporary is not introduced as a result of the use of i().)
template<class T> auto f(T)
    // #2
    -> void;
auto g() -> void {
    f(42);
    // OK: calls #2. (#1 is not a viable candidate: type deduction
    // fails (17.9.2) because A<int>::~A() is implicitly used in
    // its
    // decltype-specifier)
}

10.1.7.3 Elaborated type specifiers

elaborated-type-specifier:
    class-key attribute-specifier-seqopt nested-name-specifieropt identifier
    class-key simple-template-id
    class-key nested-name-specifier templateopt simple-template-id
    enum nested-name-specifieropt 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 (17.8.3), an explicit instantiation (17.8.2) or it has one of the following forms:

    class-key attribute-specifier-seqopt 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 templateopt 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 6.4.4 describes how name lookup proceeds for the identifier in an elaborated-type-specifier. If the identifier 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. If the identifier resolves to a typedef-name or the simple-template-id resolves to an alias template specialization, the elaborated-type-specifier is ill-formed.

[ Note: This implies that, within a class template with a template type-parameter T, the declaration

    friend class T;

is ill-formed. However, the similar declaration friend T; is allowed (14.3). — end note ]

3 The class-key or enum keyword present in the elaborated-type-specifier shall agree in kind with the declaration to which the name in the elaborated-type-specifier refers. This rule also applies to the form of elaborated-type-specifier that declares a class-name or friend class since it can be construed as referring to the definition of the class. Thus, in any elaborated-type-specifier, the enum keyword shall be used to refer to an enumeration (10.2), the union class-key shall be used to refer to a union (Clause 12), and either the class or struct class-key shall be used to refer to a class (Clause 12) declared using the class or struct class-key. [ Example:

    enum class E { a, b };
    enum E x = E::a;  // OK

§ 10.1.7.3 152

© ISO/IEC
The auto and decltype(auto) type-specifiers are used to designate a placeholder type that will be replaced later by deduction from an initializer. The auto type-specifier is also used to introduce a function type having a trailing-return-type or to signify that a lambda is a generic lambda (8.4.5). The auto type-specifier is also used to introduce a structured binding declaration (11.5).

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 (11.3.5), 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 (9.4.1).

The type of a variable declared using auto or decltype(auto) is deduced from its initializer. This use is allowed in an initializing declaration (11.6) of a variable. auto or decltype(auto) 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

\[
(\text{expression-list})
\]

the expression-list shall be a single assignment-expression. [Example:

```c
auto x = 5;  // OK: x has type int
const auto *v = &x, u = 6;  // OK: v has type const int*, u has type const int
static auto y = 0.0;  // OK: y has type double
auto int r;  // error: auto is not a storage-class-specifier
auto f() -> int;  // OK: f returns int
g() { return 0.0; }  // OK: g returns double
h();  // OK: h's return type will be deduced when it is defined
```

— end example]

A placeholder type can also be used in the type-specifier-seq in the new-type-id or type-id of a new-expression (8.5.2.4) and as a decl-specifier of the parameter-declaration’s decl-specifier-seq in a template-parameter (17.1).

A program that uses auto or decltype(auto) in a context not explicitly allowed in this subclause is ill-formed.

If the init-declarator-list contains more than one init-declarator, they shall all form declarations of variables. The type of each declared variable is determined by placeholder type deduction (10.1.7.4.1), and if the type that replaces the placeholder type is not the same in each deduction, the program is ill-formed. [Example:

```c
auto x = 5, *y = &x;  // OK: auto is int
auto a = 5, b = { 1, 2 };  // error: different types for auto
```

— end example]

If a function with a declared return type that contains a placeholder type has multiple non-discarded return statements, the return type is deduced for each such return statement. If the type deduced is not the same in each deduction, the program is ill-formed.

If a function with a declared return type that uses a placeholder type has no non-discarded return statements, the return type is deduced as though from a return statement with no operand at the closing brace of the function body. [Example:

```c
auto f() { }  // OK, return type is void
auto* g() { }  // error, cannot deduce auto* from void()
```

— end example]

If the type of an entity with an undeduced placeholder type is needed to determine the type of an expression, the program is ill-formed. Once a non-discarded return statement has been seen in a function, however, the return type deduced from that statement can be used in the rest of the function, including in other return statements. [Example:

```c
auto n = n;  // error, n's type is unknown
f();
```

— end example]
Return type deduction for a 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. \[ \textit{Note: Therefore, any use of a specialization of the function template will cause an implicit instantiation. Any errors that arise from this instantiation are not in the immediate context of the function type and can result in the program being ill-formed (17.9.2).} \] \[ \textit{—end note} \] \[ \textit{Example:} \]

\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; } // instantiates both \texttt{f}s to determine return types, // chooses second
\end{verbatim}

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. \[ \textit{Example:} \]

\begin{verbatim}
auto f();
auto f() { return 42; } // return type is int
auto f(); // OK
int f(); // error, cannot be overloaded with auto f()
dec_type(auto) f(); // error, auto and \texttt{dec_type(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 int
template char g(char); // error, no matching template
template<> auto g(double); // OK, forward declaration with unknown return type
template <class T> T g(T t) { return t; } // OK, not functionally equivalent to #1
template char g(char); // OK, now there is a matching template
template auto g(float); // still matches #1
void h() { return g(42); } // error, ambiguous
\end{verbatim}

\begin{verbatim}
template <typename T> struct A {
friend T frf(T);
};
auto frf(int i) { return i; } // not a friend of \texttt{A<int>}
\end{verbatim}

10.1.7.4.1 \textbf{Placeholder type deduction} \[ \textit{[del.type.auto.deduct]} \]

\begin{enumerate}
\item \textit{Placeholder type deduction} is the process by which a type containing a placeholder type is replaced by a deduced type.
\item A type \texttt{T} containing a placeholder type, and a corresponding initializer \texttt{e}, are determined as follows:
\end{enumerate}
(2.1) — for a non-discarded return statement that occurs in a function declared with a return type that contains a placeholder type, T is the declared return type and e is the operand of the return statement. If the return statement has no operand, then e is void();

(2.2) — for a variable declared with a type that contains a placeholder type, T is the declared type of the variable and e is the initializer. If the initialization is direct-list-initialization, the initializer shall be a braced-init-list containing only a single assignment-expression and e is the assignment-expression;

(2.3) — for a non-type template parameter declared with a type that contains a placeholder type, T is the declared type of the non-type template parameter and e is the corresponding template argument.

In the case of a return statement with no operand or with an operand of type void, T shall be either decltype(auto) or cv auto.

If the deduction is for a return statement and e is a braced-init-list (11.6.4), the program is ill-formed.

If the placeholder is the auto type-specifier, the deduced type T' replacing T is determined using the rules for template argument deduction. Obtain P from T by replacing the occurrences of auto with either a new invented type template parameter U or, if the initialization is copy-list-initialization, with std::initializer_list<U>. Deduce a value for U using the rules of template argument deduction from a function call (17.9.2.1), 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:

```cpp
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]

[Example:

```cpp
const auto &i = expr;
```

The type of i is the deduced type of the parameter u in the call f(expr) of the following invented function template:

```cpp
template <class U> void f(const U & u);
```

— end example]

If the placeholder is the decltype(auto) type-specifier, T shall be the placeholder alone. The type deduced for T is determined as described in 10.1.7.2, as though e had been the operand of the decltype. [Example:

```cpp
int i;
int* f();
auto x2a(i); // decltype(x2a) is int
dectrtype(auto) x2d(i); // decltype(x2d) is int
auto x3a = i; // decltype(x3a) is int
dectrtype(auto) x3d = i; // decltype(x3d) is int
auto x4a = {i}; // decltype(x4a) is int
dectrtype(auto) x4d = {i}; // decltype(x4d) is int*
auto x5a = f(); // decltype(x5a) is int
dectrtype(auto) x5d = f(); // decltype(x5d) is int&
auto x6a = {1, 2}; // decltype(x6a) is std::initializer_list<int>
decrtrtype(auto) x6d = {1, 2}; // error, {1, 2} is not an expression
auto *x7a = &i; // decltype(x7a) is int*
decrtrtype(auto)x7d = &i; // error, declared type is not plain decltype(auto)
```

— end example]

10.1.7.5 Deduced class template specialization types [dcl.type.class.deduct]

If a placeholder for a deduced class type appears as a decl-specifier in the decl-specifier-seq of an initializing declaration (11.6) of a variable, the placeholder is replaced by the return type of the function selected by overload resolution for class template deduction (16.3.1.8). 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.
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` (8.5.2.4), or as the `simple-type-specifier` in an explicit type conversion (functional notation) (8.5.1.3). A placeholder for a deduced class type shall not appear in any other context.

[Example:
```
template<class T> struct container {
    container(T t) {}
    template<class Iter> container(Iter beg, Iter end);
};
template<class Iter>
container(Iter b, Iter e) -> container<typename std::iterator_traits<Iter>::value_type>;
std::vector<double> v = { /* ... */ };
container c(7);  // OK, deduces int for T
auto d = container(v.begin(), v.end());  // OK, deduces double for T
container e{5, 6};  // error, int is not an iterator
```
—end example]

### 10.2 Enumeration declarations

An enumeration is a distinct type (6.7.2) with named constants. Its name becomes an `enum-name` within its scope.

```
enum-name:
  identifier

enum-specifier:
  enum-head { enumerator-list_opt }
  enum-head { enumerator-list , }

enum-head:
  enum-key attribute-specifier-seq_opt enum-head-name_opt enum-base_opt

enum-head-name:
  nested-name-specifier_opt identifier

opaque-enum-declaration:
  enum-key attribute-specifier-seq_opt nested-name-specifier_opt identifier enum-base_opt ;

enum-key:
  enum
  enum class
  enum struct

enum-base:
  : type-specifier-seq

enumerator-list:
  enumerator-definition
  enumerator-list , enumerator-definition

enumerator-definition:
  enumerator
  enumerator = constant-expression

enumerator:
  identifier attribute-specifier-seq_opt
```

The optional `attribute-specifier-seq` in the `enum-head` and the `opaque-enum-declaration` appertains to the enumeration; the attributes in that `attribute-specifier-seq` are thereafter considered attributes of the enumeration whenever it is named. A `:` following “`enum nested-name-specifier_opt identifier`” within the `decl-specifier-seq` of a `member-declaration` is parsed as part of an `enum-base`. [Note: This resolves a potential ambiguity between the declaration of an enumeration with an `enum-base` and the declaration of an unnamed bit-field of enumeration type. —Example:
```
struct S {
    enum E : int {};
    enum E : int {};  // error: redeclaration of enumeration
};
```
The enumeration type declared with an `enum-key` of only `enum` is an unscoped enumeration, and its enumerators are unscoped enumerators. The `enum-key`s `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 `identifier` 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:

```cpp
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 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.

If the `enum-key` is followed by a nested-name-specifier, the `enum-specifier` shall refer to an enumeration that was previously declared directly in the class or namespace to which the nested-name-specifier refers (i.e., neither inherited nor introduced by a using-declaration), and the `enum-specifier` shall appear in a namespace enclosing the previous declaration.

Each enumeration defines a type that is different from all other types. Each enumeration also has an 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 (8.6). If the underlying type is not fixed, the type of each `enumerator` prior to the closing brace is determined as follows:

(5.1) — If an `initializer` is specified for an `enumerator`, the `constant-expression` shall be an integral constant expression (8.6). If the expression has unscoped enumeration type, the `enumerator` has the underlying type of that enumeration type, otherwise it has the same type as the expression.

(5.2) — If no `initializer` is specified for the first `enumerator`, its type is an unspecified signed integral type.

(5.3) — Otherwise the type of the `enumerator` is the same as that of the preceding `enumerator` unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value. If no such type exists, the program is ill-formed.

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 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.

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.

For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, for an enumeration where `c_min` is the smallest `enumerator` and `c_max` is the largest,
the values of the enumeration are the values in the range \( b_{\min} \) to \( b_{\max} \), defined as follows: Let \( K \) be 1 for a two's complement representation and 0 for a one's complement or sign-magnitude representation. \( b_{\max} \) is the smallest value greater than or equal to \( \max(|e_{\min}| - K, |e_{\max}|) \) and equal to \( 2^M - 1 \), where \( M \) is a non-negative integer. \( b_{\min} \) is zero if \( e_{\min} \) is non-negative and \( -(b_{\max} + K) \) otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is \( \max(M, 1) \) if \( b_{\min} \) is zero and \( M + 1 \) otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators. If the \texttt{enumerator-list} is empty, the values of the enumeration are as if the enumeration had a single enumerator with value 0.

Two enumeration types are \textit{layout-compatible enumerations} if they have the same underlying type.

The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (7.6). [Example:
```c
enum color { red, yellow, green=20, blue };  
color col = red;  
color* cp = &col;  
if (*cp == blue)  
  // ...
```
makes \texttt{color} a type describing various colors, and then declares \texttt{col} as an object of that type, and \texttt{cp} as a pointer to an object of that type. The possible values of an object of type \texttt{color} are \texttt{red}, \texttt{yellow}, \texttt{green}, \texttt{blue}; these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type \texttt{color} can be assigned only values of type \texttt{color}.

Note that this implicit \texttt{enum} to \texttt{int} conversion is not provided for a scoped enumeration:
```c
enum class Col { red, yellow, green };  
int x = Col::red;  
Col y = Col::red;  
if (y) { }  
// error: no Col to bool conversion
```

Each \texttt{enum-name} and each unscoped \texttt{enumerator} is declared in the scope that immediately contains the \texttt{enum-specifier}. Each scoped \texttt{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:
```c
enum direction { left='l', right='r' };  

void g() {  
direction d;  
  // OK  
d = left;  
  // OK  
d = direction::right;  
  // OK
}

enum class altitude { high='h', low='l' };  

void h() {  
  altitude a;  
  // OK  
a = high;  
  // error: high not in scope  
a = altitude::low;  
  // OK
}

// end example] An enumerator declared in class scope can be referred to using the class member access operators (\texttt{::}, \texttt{.} (dot) and \texttt{->} (arrow)), see 8.5.1.5. [Example:
```c
struct X {  
  enum direction { left='l', right='r' };  
  int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p) {  
direction d;  
  // error: direction not in scope
```
int i;
i = p->f(left);        // error: left not in scope
i = p->f(X::right);    // OK
i = p->f(p->left);     // OK
// ...

—end example]  

12 If an enum-head contains a nested-name-specifier, the enum-specifier 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 (10.3.1) of that namespace (i.e., not merely inherited or introduced by a using-declaration), and the enum-specifier shall appear in a namespace enclosing the previous declaration. In such cases, the nested-name-specifier of the enum-head of the definition shall not begin with a decltype-specifier.

10.3 Namespaces

1 A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

2 The outermost declarative region of a translation unit is a namespace; see 6.3.6.

10.3.1 Namespace definition

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-namespecifier :: identifier { namespace-body }

enclosing-namespecifier:
    identifier
    enclosing-namespecifier :: 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-namespace-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-namespace-definition appears or that was introduced in a member of the inline namespace set of that namespace, the namespace-definition extends the previously-declared namespace. Otherwise, the identifier is introduced as a namespace-name into the declarative region in which the named-namespace-definition appears.

3 Because a namespace-definition contains declarations in its namespace-body and a namespace-definition is itself a declaration, it follows that namespace-definitions can be nested. [Example:

    namespace Outer {
        int i;
        namespace Inner {
            void f() { i++; }    // Outer::i
            int i;
            void g() { i++; }    // Inner::i
        }
    }

§ 10.3.1
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 10.3.1.2). Such a redeclaration has the same enclosing namespaces as the original declaration.

For example:

```cpp
namespace Q {
namespace V {
void f(); // enclosing namespaces are the global namespace, Q, and Q::V
}
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
}
}
```

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.

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` (10.3.4) that names the inline namespace is implicitly inserted into the enclosing namespace as for an unnamed namespace (10.3.1.1). Furthermore, each member of the inline namespace can subsequently be partially specialized (17.6.5), explicitly instantiated (17.8.2), or explicitly specialized (17.8.3) 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:

```cpp
namespace E { namespace I { B } }
```

For example:

```cpp
namespace A::B::C {
int i;
}
```

The above has the same effect as:

```cpp
namespace A {
namespace B {
namespace C {
int i;
}
}
}
```
10.3.1.1 Unnamed namespaces

An unnamed-namespace-definition behaves as if it were replaced by

```
inline_opt namespace unique { /* empty body */ }
using namespace unique;
namespace unique { namespace-body }
```

where inline appears if and only if it appears in the unnamed-namespace-definition and all occurrences of unique in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the translation unit. The optional attribute-specifier-seq in the unnamed-namespace-definition appertains to unique. [Example:

```cpp
namespace { int i; } // unique::i
void f() { i++; } // unique::i++

namespace A {
    namespace {
        int i; // A::unique::i
        int j; // A::unique::j
    }
    void g() { i++; } // A::unique::i++
}

using namespace A;
void h() {
    i++; // error: unique::i or A::unique::i
    A::i++; // A::unique::i
    j++; // A::unique::j
}
```

—end example]

10.3.1.2 Namespace member definitions

A declaration in a namespace `N` (excluding declarations in nested scopes) whose declarator-id is an unqualified-id (11.3), whose class-head-name (Clause 12) or enum-head-name (10.2) is an identifier, or whose elaborated-type-specifier is of the form class-key attribute-specifier-seq_opt identifier (10.1.7.3), or that is an opaque_enum_declaration, declares (or redeclares) its unqualified-id or identifier as a member of `N`. [Note: An explicit instantiation (17.8.2) or explicit specialization (17.8.3) 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
}
```
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:

// 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
    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]

10.3.2 Namespace alias [namespace.alias]

1 A namespace-alias-definition declares an alternate name for a namespace according to the following grammar:

    namespace-alias:
      identifier

    namespace-alias-definition:
      namespace identifier = qualified-name-specifier ;

    qualified-name-specifier:
      nested-name-specifier_opt namespace-name

99) this implies that the name of the class or function is unqualified.
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]

In a declarative region, a namespace-alias-definition can be used to redefine a namespace-alias declared in that declarative region to refer only to the namespace to which it already refers. [Example: The following declarations are well-formed:

```
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name; // OK: duplicate
namespace CWVLN = CWVLN;
```
—end example]

10.3.3 The using declaration

```
using-declaration:
  using using-declarator-list ;
using-declarator-list:
  using-declarator ...opt
  using-declarator-list , using-declarator ...opt
using-declarator:
  typenameopt nested-name-specifier unqualified-id
```

Each using-declarator in a using-declaration\(^\text{100}\) 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, 13.2) 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:

```
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:

```
template <typename... bases>
struct X : bases... {
  using bases::g...;
};
X<B, D> x; // OK: B::g and D::g introduced
```
\(^{100}\) A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using-declarator each.
Example:

class C {
    int g();
};

class D2 : public B {
    using B::f;
    using B::e;
    using B::x;
    using C::g;
};

—end example

Note: Since destructors do not have names, a using-declaration cannot refer to a destructor for a base class. Since specializations of member templates for conversion functions are not found by name lookup, they are not considered when a using-declaration specifies a conversion function (17.6.2). —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 (15.8), 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.

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]

A using-declaration shall not name a namespace.

A using-declaration shall not name a scoped enumerator.

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]

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
}
void h()
{
    X::f(); // calls ::f
    X::g(); // calls A::g
}
— end example]

A using-declaration is a declaration and can therefore be used repeatedly where (and only where) multiple
declarations are allowed. [Example:

namespace A {
    int i;
}

namespace A1 {
    using A::i, A::i; // OK: double declaration
}

struct B {
    int i;
};

struct X : B {
    using B::i, B::i; // error: double member declaration
};
— end example]

[Note: For a using-declaration whose nested-name-specifier names a namespace, members added to the
namespace after the using-declaration are not in the set of introduced declarations, so they are not considered
when a use of the name is made. Thus, additional overloads added after the using-declaration are ignored, but
default function arguments (11.3.6), default template arguments (17.1), and template specializations (17.6.5,
17.8.3) are considered. — end note] [Example:

namespace A {
    void f(int);
}

using A::f; // f is a synonym for A::f; that is, for A::f(int).
namespace A {
    void f(char);
}

void foo() {
    f('a'); // calls f(int), even though f(char) exists.
}

void bar() {
    using A::f; // f is a synonym for A::f; that is, for A::f(int) and A::f(char).
    f('a'); // calls f(char)
}
— end example]

[Note: 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 (17.6.5). — 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:

namespace A {
    int x;
}
namespace B {
  int i;
  struct g { }; // OK: hides struct g
  struct x { }
  void f(int);
  void f(double);
  void g(char);
}

void func() {
  int i;
  using B::i; // error: i declared twice
  void f(char);
  using B::f; // OK: each f is a function
  f(3.5); // calls B::f(double)
  using B::g;
  g('a'); // calls B::g(char)
  struct g g1; // g1 has class type B::g
  using B::x;
  using A::x; // OK: hides struct B::x
  x = 99; // assigns to A::x
  struct x x1; // x1 has class type B::x
}

—end example—

If a function declaration in namespace scope or block scope has the same name and the same parameter-type-list (11.3.5) 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 (11.3.5), 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);
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]
18 [Note: Because a *using-declarator* designates a base class member (and not a member subobject or a member function of a base class subobject), a *using-declarator* cannot be used to resolve inherited member ambiguities.]

[Example:

```cpp
struct A { int x(); }
struct B : A { }
struct C : A {
    using A::x;
    int x(int);
};

struct D : B, C {
    using C::x;
    int x(double);
};
int f(D* d) {
    return d->x(); // error: overload resolution selects A::x, but A is an ambiguous base class
}
```]

—end example] —end note]

A synonym created by a *using-declaration* has the usual accessibility for a *member-declaration*. A *using-declarator* that names a constructor does not create a synonym; instead, the additional constructors are accessible if they would be accessible when used to construct an object of the corresponding base class, and the accessibility of the *using-declaration* is ignored. [Example:

```cpp
class A {
    private:
        void f(char);
    public:
        void f(int);
    protected:
        void g();
};

class B : public A {
    using A::f; // error: A::f(char) is inaccessible
    public:
        using A::g; // B::g is a public synonym for A::g
    }
```]

—end example]

19 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:

```cpp
class A {
    private:
        void f(char);
    public:
        void f(int);
    protected:
        void g();
};

class B : public A {
    using A::f; // error: A::f(char) is inaccessible
    public:
        using A::g; // B::g is a public synonym for A::g
    }
```]

—end example]

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:

```cpp
class A {
    private:
        void f(char);
    public:
        void f(int);
    protected:
        void g();
};

class B : public A {
    using A::f; // error: A::f(char) is inaccessible
    public:
        using A::g; // B::g is a public synonym for A::g
    }
```]

—end example]

10.3.4 Using directive

```cpp
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:

```cpp
namespace A {
    int i;
}
namespace B {
    namespace C {
        int i;
    }
```]

§ 10.3.4

168
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:
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,
namespace A {
    int i;
}
namespace B {
    int i;
    int j;
    namespace C {
        namespace D {
            using namespace A;
            int j;
            int k;
            int a = i; // B::i hides A::i
        }
        using namespace D;
        int k = 89; // no problem yet
        int l = k; // ambiguous: C::k or D::k
        int m = i; // B::i hides A::i
        int n = j; // D::j hides B::j
    }
}
—end example]

§ 10.3.4
If a namespace is extended (10.3.1) 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.

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, the use of the name is ill-formed. [Note: 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]
```

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:]

```cpp
namespace D {
    int d1;
    void f(char);
}
using namespace D;

int d1; // OK: no conflict with D::d1

namespace E {
    int e;
    void f(int);
}

namespace D { // namespace extension
    int d2;
    using namespace E;
    void f(int);
}

void f() {
    d1++; // error: ambiguous ::d1 or D::d1?
    ::d1++; // OK
    D::d1++; // OK
    d2++; // OK: D::d2
    e++; // OK: E::e
}
101) During name lookup in a class hierarchy, some ambiguities may be resolved by considering whether one member hides the other along some paths (13.2). There is no such disambiguation when considering the set of names found as a result of following using-directives.

§ 10.3.4
10.4 The \texttt{asm} declaration \[dcl.asm]\n
An \texttt{asm} declaration has the form

\begin{verbatim}
asm-definition:
  attribute-specifier-seq_opt asm ( string-literal ) ;
\end{verbatim}

The \texttt{asm} declaration is conditionally-supported; its meaning is implementation-defined. The optional \texttt{attribute-specifier-seq} in an \texttt{asm-definition} appertains to the \texttt{asm} declaration. [\textit{Note:} Typically it is used to pass information through the implementation to an assembler. — end note]

10.5 Linkage specifications \[dcl.link]\n
All function types, function names with external linkage, and variable names with external linkage have a language linkage. [\textit{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.

Linkage (6.5) between C++ and non-C++ code fragments can be achieved using a \texttt{linkage-specification}:

\begin{verbatim}
linkage-specification:
  extern string-literal \{ declaration-seq_opt \}
  extern string-literal declaration
\end{verbatim}

The \texttt{string-literal} indicates the required language linkage. This document specifies the semantics for the \texttt{string-liters "C" and "C++".} Use of a \texttt{string-literal} other than \texttt{"C"} or \texttt{"C++"} is conditionally-supported, with implementation-defined semantics. [\textit{Note:} Therefore, a linkage-specification with a \texttt{string-literal} that is unknown to the implementation requires a diagnostic. — end note] [\textit{Note:} It is recommended that the spelling of the \texttt{string-literal} be taken from the document defining that language. For example, \texttt{Ada} (not \texttt{ADA}) and \texttt{Fortran} or \texttt{FORTRAN}, depending on the vintage. — end note]

Every implementation shall provide for linkage to functions written in the C programming language, \texttt{"C"}, and linkage to C++ functions, \texttt{"C++"}. [\textit{Example:}

\begin{verbatim}
complex sqrt(complex); // C++ linkage by default
extern "C" {
  double sqrt(double); // C linkage
}
\end{verbatim}

— 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 \texttt{linkage-specification} shall occur only in namespace scope (6.3). In a \texttt{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 \texttt{linkage-specification}. [\textit{Example:}

\begin{verbatim}
extern "C" // the name \texttt{f1} and its function type have C language linkage;
void f1(void(*pf)(int)); // \texttt{pf} is a pointer to a C function

extern "C" typedef void FUNCTION();
FUNCTION f2; // the name \texttt{f2} has C++ language linkage and the
  // function's type has C language linkage

extern "C" FUNCTION f3; // the name of function \texttt{f3} and the function's type have C language linkage

void (*pf2)(FUNCTION*); // \texttt{pf2} is “pointer to C++ function that takes one parameter of type
  // pointer to C function”

extern "C" {
  static void f4(); // the name of the function \texttt{f4} has internal linkage (not C language linkage)
  
  \texttt{extern "C"} { // <- this \texttt{extern "C"} is ignored
    void f5(); // bad linkage
  }

  \texttt{extern "C"} typedef void FUNCTION();
  FUNCTION f6; // bad linkage

  \texttt{extern "C"} { // <- another ignored \texttt{extern "C"}
    void f7(); // bad linkage
  }

  \texttt{extern "C"} {
    static void f8(); // bad linkage
  }

  \texttt{extern "C"} {
    \texttt{extern "C"} {
      void f9(); // bad linkage
    }
  }

  \texttt{extern "C"} {
    \texttt{extern "C"} {
      \texttt{extern "C"} {
        \texttt{extern "C"} {
          \texttt{extern "C"} {
            \texttt{extern "C"} {
              \texttt{extern "C"} {
                \texttt{extern "C"} {
                  \texttt{extern "C"} {
                    \texttt{extern "C"} {
                      \texttt{extern "C"} {
                        \texttt{extern "C"} {
                          \texttt{extern "C"} {
                            \texttt{extern "C"} { // <- another ignored \texttt{extern "C"}
                              void f10(); // bad linkage
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
\end{verbatim}

— end example]
/* and the function’s type has C language linkage. */

extern "C" void f5() {
extern void f4(); // OK: Name linkage (internal) and function type linkage (C language linkage)
// obtained from previous declaration.
}

extern void f4(); // OK: Name linkage (internal) and function type linkage (C language linkage)
// obtained from previous declaration.

void f6() {
extern void f4(); // OK: Name linkage (internal) and function type linkage (C language linkage)
// obtained from previous declaration.
}

—end example] A C language linkage is ignored in determining the language linkage of the names of class members and the function type of class member functions. [Example:

extern "C" typedef void FUNC_c();

class C {
void mf1(FUNC_c*);
// the name of the function mf1 and the member function’s type have
// C++ language linkage; the parameter has type “pointer to C function”

FUNC_c mf2; // the name of the function mf2 and the member function’s type have
// C++ language linkage

static FUNC_c* q; // the name of the data member q has C++ language linkage and
// the data member’s type is “pointer to C function”
};

extern "C" {
class X {
void mf(); // the name of the function mf and the member function’s type have
// C++ language linkage

void mf2(void(*)()); // the name of the function mf2 has C++ language linkage;
// the parameter has type “pointer to C function”
};
}

—end example]

5 If two declarations declare functions with the same name and parameter-type-list (11.3.5) 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:

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; }   // A::h and ::h refer to the same function
— end example]

A declaration directly contained in a linkage-specification is treated as if it contains the extern specifier (10.1.1) 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:

    extern "C" double f();
    static double f();   // error
    extern "C" int i;   // declaration
    extern "C"
        int i;   // definition
    
    extern "C" static void g();   // error
    — end example]

[ Note: Because the language linkage is part of a function type, when indirecting through a pointer to C function, the function to which the resulting lvalue refers is considered a C function. — end note ]

Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

10.6 Attributes

10.6.1 Attribute syntax and semantics

Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

attribute-specifier-seq:
  attribute-specifier-seq_opt attribute-specifier

attribute-specifier:
  [[ attribute-using-prefix_opt attribute-list ]]
  alignment-specifier

alignment-specifier:
  alignas ( type-id ... opt )
  alignas ( constant-expression ... opt )

attribute-using-prefix:
  using attribute-namespaces :

attribute-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

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:

```c
[[using CC:: opt(1), debug]] // same as [[CC::opt(1), CC::debug]]
void f() {}
[[using CC:: opt(1)]] [[CC::debug]] // same as [[CC::opt(1)]] [[CC::debug]]
void g() {}
[[using CC:: CC::opt(1)]] // error: cannot combine using and scoped attribute token
void h() {}
```

—end example]

3 [Note: For each individual attribute, the form of the balanced-token-seq will be specified. —end note]

4 In an attribute-list, an ellipsis may appear only if that attribute’s specification permits it. An attribute followed by an ellipsis is a pack expansion (17.6.3). 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 9, Clause 10, Clause 11). 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 (14.3), that declaration shall be a definition. No attribute-specifier-seq shall appertain to an explicit instantiation (17.8.2).

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] [Example:

```c
int p[10];
void f() {
  int x = 42, y[5];
  int p[10] { return x; }(); // error: invalid attribute on a nested declarator-id and
  // not a function-style cast of an element of p.
  y[1] { return 2; }(); = 2; // error even though attributes are not allowed in this context.
  int i [[vendor::attr([])]]; // well-formed implementation-defined attribute.
}
```

—end example]
10.6.2 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 (18.3). An alignment-specifier may also be applied to the declaration or definition of a class (in an elaborated-type-specifier (10.1.7.3) or class-head (Clause 12), respectively) and to the declaration or definition of an enumeration (in an opaque-enum-declaration or enum-head, respectively (10.2)). An alignment-specifier with an ellipsis is a pack expansion (17.6.3).

2 When the alignment-specifier is of the form alignas( constant-expression ):

(2.1) when the constant-expression shall be an integral constant expression
(2.2) if the constant expression does not evaluate to an alignment value (6.6.5), or evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed.

3 An alignment-specifier of the form alignas( type-id ) has the same effect as alignas(alignof( type-id )) (8.5.2.6).

4 The alignment requirement of an entity is the strictest nonzero alignment specified by its alignment-specifiers, if any; otherwise, the alignment-specifiers have no effect.

5 The combined effect of all alignment-specifiers in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all alignment-specifiers appertaining to that entity were omitted. [Example:

```c
struct alignas(8) S {};
struct alignas(1) U {
    S s;
}; // error: U specifies an alignment that is less strict than if the alignas(1) were omitted.
```

—end example]

6 If the defining declaration of an entity has an alignment-specifier, any non-defining declaration of that entity shall either specify equivalent alignment or have no alignment-specifier. Conversely, if any declaration of an entity has an alignment-specifier, every defining declaration of that entity shall specify an equivalent alignment. No diagnostic is required if declarations of an entity have different alignment-specifiers in different translation units. [Example:

```c
// Translation unit #1:
struct S { int x; } s, *p = &s;

// Translation unit #2:
struct alignas(16) S; // error: definition of S lacks alignment, no diagnostic required
extern S* p;
```

—end example]

7 [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]

8 [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]

10.6.3 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-value
conversion (7.1) of that object. The attribute may also be applied to the \textit{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 \texttt{carries\_dependency} attribute for its \texttt{declarator-id} if any declaration of the function specifies the \texttt{carries\_dependency} attribute. Furthermore, the first declaration of a function shall specify the \texttt{carries\_dependency} attribute for a parameter if any declaration of that function specifies the \texttt{carries\_dependency} attribute for that parameter. If a function or one of its parameters is declared with the \texttt{carries\_dependency} attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the \texttt{carries\_dependency} attribute in its first declaration in another translation unit, the program is ill-formed, no diagnostic required.

\texttt{carries\_dependency} attribute does not change the meaning of the program, but may result in generation of more efficient code. — end note

\textbf{Example:}

\begin{verbatim}
/* Translation unit A. */

struct foo { int* a; int* b; };
std::atomic<struct foo *> foo_head[10];
int foo_array[10][10];

[[carries\_dependency]] struct foo* f(int i) {
  return foo_head[i].load(memory_order::consume);
}

int g(int* x, int* y [[carries\_dependency]]) {
  return kill_dependency(foo_array[*x][*y]);
}

/* Translation unit B. */

[[carries\_dependency]] struct foo* f(int i);
int g(int* x, int* y [[carries\_dependency]]);

int c = 3;

void h(int i) {
  struct foo* p;

  p = f(i);
  do_something_with(g(&c, p->a));
  do_something_with(g(p->a, &c));
}
\end{verbatim}

The \texttt{carries\_dependency} attribute on function \texttt{f} means that the return value carries a dependency out of \texttt{f}, so that the implementation need not constrain ordering upon return from \texttt{f}. Implementations of \texttt{f} and its caller may choose to preserve dependencies instead of emitting hardware memory ordering instructions (a.k.a. fences). Function \texttt{g}'s second parameter has a \texttt{carries\_dependency} attribute, but its first parameter does not. Therefore, function \texttt{h}'s first call to \texttt{g} carries a dependency into \texttt{g}, but its second call does not. The implementation might need to insert a fence prior to the second call to \texttt{g}. — end example]

\section*{10.6.4 Deprecated attribute [dcl.attr.deprecated]}

The \texttt{attribute-token} \texttt{deprecated} can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. [\textit{Note: In particular, \texttt{deprecated} is appropriate for names and entities that are deemed obsolescent or unsafe. — end note}] It shall appear at most once in each \texttt{attribute-list}. An \texttt{attribute-argument-clause} may be present and, if present, it shall have the form:

\begin{verbatim}
(string-literal)
\end{verbatim}

[\textit{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 \texttt{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]

### 10.6.5 Fallthrough attribute

The attribute-token fallthrough may be applied to a null statement (9.2); 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 (9.4.2). 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]];
        case 3:    // warning on fallthrough discouraged
            h();
        case 4:    // implementation may warn on fallthrough
            i();
            [[fallthrough]];  // ill-formed
    }  
}
```

— end example]

### 10.6.6 Maybe unused attribute

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.

The attribute may be applied to the declaration of a class, a typedef-name, a variable, a non-static data member, a function, an enumeration, or an enumerator.

[Note: For an entity marked maybe_unused, implementations should not emit a warning that the entity is unused, or that the entity is used despite the presence of the attribute. — end note]

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.

[Example:

```c
[[maybe_unused]] void f([[maybe_unused]] bool thing1,
    [[maybe_unused]] bool thing2) {
    [[maybe_unused]] bool b = thing1 && thing2;
    assert(b);  
}
```

Implementations should not warn that b is unused, whether or not NDEBUG is defined. — end example]
10.6.7 Nodiscard attribute

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.

[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 (8.2) 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]

Example:

```
struct [[nodiscard]] error_info { /* ... */ };
error_info enable_missile_safety_mode();
void launch_missiles();
void test_missiles()
    { enable_missile_safety_mode(); // warning encouraged
      launch_missiles();
    }
error_info &foo();
void f() { foo(); } // warning not encouraged: not a nodiscard call, because neither
          // the (reference) return type nor the function is declared nodiscard
```

—end example]

10.6.8 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.

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]

Example:

```
[[ noreturn ]] void f() {
    throw "error";    // OK
}

[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
    if (i > 0)
        throw "positive";
}
```

—end example]
11 Declarators

A declarator declares a single variable, function, or type, within a declaration. The \texttt{init-declarator-list} appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

\texttt{init-declarator-list:}
\begin{itemize}
  \item \texttt{init-declarator}
  \item \texttt{init-declarator-list, init-declarator}
\end{itemize}

\texttt{init-declarator:}
\begin{itemize}
  \item \texttt{declarator initializer_{opt}}
  \item \texttt{declarator requires-clause}
\end{itemize}

The three components of a \texttt{simple-declaration} are the attributes (10.6), the specifiers (\texttt{decl-specifier-seq}; 10.1) and the declarators (\texttt{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 \texttt{*} (pointer to) and \texttt{()} (function returning). Initial values can also be specified in a declarator; initializers are discussed in 11.6 and 15.6.

Each \texttt{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
\begin{equation*}
T D_1, D_2, \ldots, D_n;
\end{equation*}
is usually equivalent to
\begin{equation*}
T D_1; T D_2; \ldots T D_n;
\end{equation*}
where \texttt{T} is a \texttt{decl-specifier-seq} and each \texttt{D_i} is an \texttt{init-declarator}. One exception is when a name introduced by one of the \texttt{declarators} hides a type name used by the \texttt{decl-specifiers}, so that when the same \texttt{decl-specifiers} are used in a subsequent declaration, they do not have the same meaning, as in
\begin{equation*}
\text{struct } S \{ \ldots \};
\end{equation*}
\begin{equation*}
S S, T; \quad \text{// declare two instances of struct } S
\end{equation*}
which is not equivalent to
\begin{equation*}
\text{struct } S \{ \ldots \};
\end{equation*}
\begin{equation*}
S S;
\end{equation*}
\begin{equation*}
S T; \quad \text{// error}
\end{equation*}
Another exception is when \texttt{T} is \texttt{auto} (10.1.7.4), for example:
\begin{equation*}
\text{auto } i = 1, j = 2.0; \quad \text{// error: deduced types for } i \text{ and } j \text{ do not match}
\end{equation*}
as opposed to
\begin{equation*}
\text{auto } i = 1; \quad \text{// OK: } i \text{ deduced to have type } \text{int}
\end{equation*}
\begin{equation*}
\text{auto } j = 2.0; \quad \text{// OK: } j \text{ deduced to have type } \text{double}
\end{equation*}
\hspace{1cm} — end note]

The optional \texttt{requires-clause} (Clause 17) in an \texttt{init-declarator} or \texttt{member-declarator} shall not be present when the declarator does not declare a function (11.3.5). When present after a declarator, the \texttt{requires-clause} is called the \texttt{trailing requires-clause}. The trailing \texttt{requires-clause} introduces the \texttt{constraint-expression} that results from interpreting its \texttt{constraint-logical-or-expression} as a \texttt{constraint-expression}. [Example:
\begin{equation*}
\text{void } f1(int a) \text{ requires true; \quad // OK}
\end{equation*}
\begin{equation*}
\text{auto } f2(int a) \rightarrow \text{bool requires true; \quad // OK}
\end{equation*}
\begin{equation*}
\text{auto } f3(int a) \text{ requires true } \rightarrow \text{bool; \quad // error: requires-clause precedes } \text{trailing-return-type}
\end{equation*}
\begin{equation*}
\text{void } (*pf)() \text{ requires true; \quad // error: constraint on a variable}
\end{equation*}
\begin{equation*}
\text{void } g(int (*)(\star)()) \text{ requires true; \quad // error: constraint on a } \text{parameter-declaration}
\end{equation*}
\begin{equation*}
\text{auto* } p = \text{new } \text{void(*)(char) requires true; \quad // error: not a function declaration}
\end{equation*}
\hspace{1cm} — end example]

Declarators have the syntax
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
  noptr-declarator parameters-and-qualifiers
  noptr-declarator [ constant-expression ] attribute-specifier-seq
  ( ptr-declarator )

parameters-and-qualifiers:
  ( parameter-declaration-clause ) cv-qualifier-seq
  ref-qualifier_opt noexcept-specifier_opt attribute-specifier-seq

trailing-return-type:
  -> type-id

ptr-operator:
  * attribute-specifier-seq cv-qualifier-seq
  & attribute-specifier-seq
  && attribute-specifier-seq
  nested-name-specifier * attribute-specifier-seq cv-qualifier-seq

cv-qualifier-seq:
  cv-qualifier cv-qualifier-seq


cv-qualifier:
  const
  volatile

ref-qualifier:
  &
  &&

declarator-id:
  . . . opt id-expression

11.1 Type names

To specify type conversions explicitly, and as an argument of sizeof, alignof, new, or typeid, the name of a type shall be specified. This can be done with a type-id, which is syntactically a declaration for a variable or function of that type that omits the name of the entity.

type-id:
  type-specifier-seq abstract-declarator

defining-type-id:
  defining-type-specifier-seq abstract-declarator

abstract-declarator:
  ptr-abstract-declarator
  noptr-abstract-declarator parameters-and-qualifiers trailing-return-type
  abstract-pack-declarator

ptr-abstract-declarator:
  noptr-abstract-declarator
  ptr-operator ptr-abstract-declarator

noptr-abstract-declarator:
  noptr-abstract-declarator parameters-and-qualifiers
  noptr-abstract-declarator [ constant-expression ] attribute-specifier-seq
  ( ptr-abstract-declarator )

abstract-pack-declarator:
  noptr-abstract-pack-declarator
  ptr-operator abstract-pack-declarator

[ dcl.name ]
It is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. [Example:

```
int // int i
int * // int *pi
int *[3] // int *[p][3]
int (*[3]) // int (*)(*p3i)[3]
int *() // int *f()
int (*)(double) // int (*)(*p)(double)
```

name respectively the types “int”, “pointer to int”, “array of 3 pointers to int”, “pointer to array of 3 int”, “function of (no parameters) returning pointer to int”, and “pointer to a function of (double) returning int”. —end example]

A type can also be named (often more easily) by using a typedef (10.1.3).

### 11.2 Ambiguity resolution

The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 9.8 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 9.8, 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:

```
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]

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:

```
template <class T> struct X {};
template <int N> struct Y {};
X<int()> a; // type-id
X<int()> b; // expression (ill-formed)
Y<int()> c; // type-id (ill-formed)
Y<int()> d; // expression

void foo(signed char a) {
    sizeof(int()); // type-id (ill-formed)
    sizeof(int(a)); // expression
    sizeof(int(unsigned(a))); // type-id (ill-formed)
    (int())+1; // type-id (ill-formed)
    (int(a))+1; // expression
    (int(unsigned(a))+1; // type-id (ill-formed)
}

—end example]

§ 11.2 181
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(C)) { }  // not: void f(int C) {
// void f(int(*fp)(C c)) { }
// not: void f(int C) {
int g(C);
// void foo() {
// error: cannot convert 1 to function pointer
void foo() {
    f(1);  // OK
    f(g);
}
// not: void h(int C[10]);
For another example,
```n
class C { };
void h(int *(C[10]));  // void h(int *(fp)(C _parm[10]));
// not: void h(int *C[10]);

—end example]

### 11.3 Meaning of declarators

A declarator contains exactly one declarator-id; it names the identifier that is declared. An unqualified-id occurring in a declarator-id shall be a simple identifier except for the declaration of some special functions (15.1, 15.3, 15.4, 16.5) and for the declaration of template specializations or partial specializations (17.8). 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 (10.3.1)) 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.

A static, thread_local, extern, mutable, friend, inline, virtual, constexpr, explicit, or typedef specifier applies directly to each declarator-id in an init-declarator-list or member-declarator-list; the type specified for each declarator-id depends on both the decl-specifier-seq and its declarator.

Thus, a declaration of a particular identifier has the form

```
T D
```

where T is of the form attribute-specifier-seq opt decl-specifier-seq and D is a declarator. Following is a recursive procedure for determining the type specified for the contained declarator-id by such a declaration.

First, the decl-specifier-seq determines a type. In a declaration

```
T D
```

the decl-specifier-seq T determines the type T. [Example: In the declaration

```c
int unsigned i;
```

the type specifiers int unsigned determine the type “unsigned int” (10.1.7.2). — end example]

In a declaration attribute-specifier-seq opt T D where D is an unadorned identifier the type of this identifier is “T”.

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.
11.3.1 Pointers

In a declaration \( T D \) where \( D \) has the form

\[
* \text{attribute-specifier-seq}_\text{opt} \text{ cv-qualifier-seq}_\text{opt} D1
\]

and the type of the identifier in the declaration \( T D1 \) is “derived-declarator-type-list \( T \)”, then the type of the identifier of \( D \) is “derived-declarator-type-list cv-qualifier-seq pointer to \( T \)”. The cv-qualifiers apply to the pointer and not to the object pointed to. Similarly, the optional attribute-specifier-seq (10.6.1) appertains to the pointer and not to the object pointed to.

Example: The declarations

\[
\begin{align*}
\text{const int } ci &= 10, *pc = &ci, *\text{const cpc} = pc, **ppc; \\
\text{int } i, *p, *\text{const cp} = &i;
\end{align*}
\]
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

\[
\begin{align*}
i &= ci; \\
*cp &= ci; \\
p &= pc; \\
ppc &= &pc;
\end{align*}
\]

Examples of ill-formed operations are

\[
\begin{align*}
\text{ci } &= 1; & \text{error} \\
\text{ci} &= \text{++}; & \text{error} \\
*\text{pc } &= 2; & \text{error} \\
\text{cp } &= &\text{ci}; & \text{error} \\
\text{cpc} &= \text{++}; & \text{error} \\
\text{p } &= \text{pc}; & \text{error} \\
\text{ppc } &= &\text{p}; & \text{error}
\end{align*}
\]

Each is unacceptable because it would either change the value of an object declared \texttt{const} or allow it to be changed through a cv-unqualified pointer later, for example:

\[
\begin{align*}
*\text{ppc } &= &\text{ci}; & \text{OK, but would make } p \text{ point to } ci \text{ because of previous error} \\
*\text{p } &= 5; & \text{error} \quad \text{clobber } ci
\end{align*}
\]

—end example]

Note: Forming a pointer to reference type is ill-formed; see 11.3.2. Forming a function pointer type is ill-formed if the function type has cv-qualifiers or a ref-qualifier; see 11.3.5. Since the address of a bit-field (12.2.4) cannot be taken, a pointer can never point to a bit-field. —end note

11.3.2 References

In a declaration \( T D \) where \( D \) has either of the forms

\[
& \text{attribute-specifier-seq}_\text{opt} D1 \\
&\& \text{attribute-specifier-seq}_\text{opt} D1
\]

and the type of the identifier in the declaration \( T D1 \) is “derived-declarator-type-list \( T \)”, then the type of the identifier of \( D \) is “derived-declarator-type-list reference to \( T \)”. The optional attribute-specifier-seq appertains to the reference type. Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef-name (10.1.3, 17.1) or decltype-specifier (10.1.7.2), in which case the cv-qualifiers are ignored. [Example:

\[
\begin{align*}
\text{typedef int& A;} \\
\text{const A aref } &= 3; & \text{ill-formed; lvalue reference to non-const initialized with rvalue}
\end{align*}
\]

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.

[Example:

```c
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```
declares a to be a reference parameter of f so the call f(d) will add 3.14 to d.

```c
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```
declares the function g() to return a reference to an integer so g(3)=7 will assign 7 to the fourth element of the array v. For another example,

```c
struct link {
    link* next;
};
link* first;

void h(link*& p) {
    // p is a reference to pointer
    p->next = first;
    first = p;
    p = 0;
}

void k() {
    link* q = new link;
h(q);
}
```
declares p to be a reference to a pointer to link so h(q) will leave q with the value zero. See also 11.6.3.

—end example]

It is unspecified whether or not a reference requires storage (6.6.4).

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 (11.6.3) except when the declaration contains an explicit extern specifier (10.1.1), is a class member (12.2) declaration within a class definition, or is the declaration of a parameter or a return type (11.3.5); 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 12.2.4, a reference cannot be bound directly to a bit-field. —end note]

If a typedef-name (10.1.3, 17.1) or a decltype-specifier (10.1.7.2) 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&&
```
decltype(r2)& r6 = i; // r6 has the type int&
decltype(r2)&& r7 = i; // r7 has the type int&

— end example]

[ Note: Forming a reference to function type is ill-formed if the function type has cv-qualifiers or a ref-qualifier; see 11.3.5. — end note]

11.3.3 Pointers to members

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 (10.6.1) appertains to the pointer-to-member.

2 [ Example:

    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]

3 A pointer to member shall not point to a static member of a class (12.2.3), a member with reference type, or “cv void”.

[ Note: See also 8.5.2 and 8.5.4. 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]

11.3.4 Arrays

1 In a declaration T D where D has the form

    D1 [ constant-expression_opt ] attribute-specifier-seq_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, a function type or an abstract class type. If the constant-expression (8.6) 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-qualified-seq array of N T” is adjusted to “array of N cv-qualified-seq T”, and similarly for “array of unknown bound of T”. The optional attribute-specifier-seq appertains to the array. [Example:

typedef int A[5], AA[2][3];  
typedef const A CA;          // type is “array of 5 const int”  
typedef const AA CAA;        // type is “array of 2 array of 3 const int”
—end example]  [Note: An “array of N cv-qualified-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 (11.3.5). An array bound may also be omitted when the declarator is followed by an initializer (11.6) or when a declarator for a static data member is followed by a brace-or-equal-initializer (12.2). In both cases the bound is calculated from the number of initial elements (say, N) supplied (11.6.1), 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:

float fa[17], *afp[17];
declares an array of float numbers and an array of pointers to float numbers. —end example]

5 [Example:

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 × 5 × 7. Any of the expressions x3d, x3d[1], 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 (7.2) and is first converted to a pointer to a 2-dimensional array with rank 5 × 7 that points to the first element of x3d. Then i is added, which on typical implementations involves multiplying i by the length of the object to which the pointer points, which is sizeof(int)×5 × 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]

6 [Example:

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]

7 [Note: Conversions affecting expressions of array type are described in 7.2. Objects of array types cannot be modified, see 8.2.1. —end note]

8 [Note: Except where it has been declared for a class (16.5.5), the subscript operator [] is interpreted in such a way that E1[E2] is identical to *((E1)+(E2)) (8.5.1.1). Because of the conversion rules that apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation. —end note]

§ 11.3.4
11.3.5 Functions

1 In a declaration $T D$ where $D$ has the form

$$D1 \ ( \ parameter-declaration-clause \ ) \ cv-qualifier-seqopt \ ref-qualifier opt noexcept-specifier opt attribute-specifier-seqopt$$

and the type of the contained declarator-id in the declaration $T D1$ is “derived-declarator-type-list $T$”, the type of the declarator-id in $D$ is “derived-declarator-type-list noexcept opt function of (parameter-declaration-clause) cv-qualifier-seqopt ref-qualifier opt returning $T$”, where the optional noexcept is present if and only if the exception specification (18.4) is non-throwing. The optional attribute-specifier-seq appertains to the function type.

2 In a declaration $T D$ where $D$ has the form

$$D1 \ ( \ parameter-declaration-clause \ ) \ cv-qualifier-seqopt \ ref-qualifier opt noexcept-specifier opt attribute-specifier-seqopt trailing-return-type$$

and the type of the contained declarator-id in the declaration $T D1$ is “derived-declarator-type-list $T$”, $T$ shall be the single type-specifier auto. The type of the declarator-id in $D$ is “derived-declarator-type-list noexcept opt function of (parameter-declaration-clause) cv-qualifier-seqopt 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.102

$$\begin{align*}
& parameter-declaration-clause: \\
& \quad parameter-declaration-list_{opt} \ldots_{opt} \\
& \quad parameter-declaration-list, \ldots \\
& parameter-declaration-list: \\
& \quad parameter-declaration \\
& \quad parameter-declaration-list, \ parameter-declaration \\
& parameter-declaration: \\
& \quad attribute-specifier-seq_{opt} \ decl-specifier-seq \ declarator \\
& \quad attribute-specifier-seq_{opt} \ decl-specifier-seq \ declarator = initializer-clause \\
& \quad attribute-specifier-seq_{opt} \ decl-specifier-seq abstract-declarator_{opt} \\
& \quad attribute-specifier-seq_{opt} \ decl-specifier-seq abstract-declarator_{opt} = initializer-clause
\end{align*}$$

The optional attribute-specifier-seq in a parameter-declaration appertains to the parameter.

4 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 8.5.1.2. — 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 (17.6.3), 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

```c
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```c
printf("hello world");
printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a const char* — end example] [Note: The standard header <stdarg.h> contains a mechanism for accessing arguments passed using the ellipsis (see 8.5.1.2 and 21.11). — end note]

5 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.

---

102) As indicated by syntax, cv-qualifiers are a significant component in function return types.
A function type with a `cv-qualifier-seq` or a `ref-qualifier` (including a type named by `typedef-name` (10.1.3, 17.1)) shall appear only as:

- the function type for a non-static member function,
- the function type to which a pointer to member refers,
- the top-level function type of a function typedef declaration or `alias-declaration`,
- the `type-id` in the default argument of a `type-parameter` (17.1), or
- the `type-id` of a `template-argument` for a `type-parameter` (17.3.1).

```cpp
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
```

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]  

```cpp
typedef void F();
struct S {
    const F f;
    // OK: equivalent to: void f();
};
```

The return type, the parameter-type-list, the `ref-qualifier`, the `cv-qualifier-seq`, and the exception specification, but not the default arguments (11.3.6) or the trailing `requires-clause` (Clause 11), 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]

```cpp
int fseek(FILE*, long, int);
```

A single name can be used for several different functions in a single scope; this is function overloading (Clause 16). All declarations for a function shall have equivalent return types, parameter-type-lists, and `requires-clauses` (17.6.6.1).

Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions.

Types shall not be defined in return or parameter types. The type of a parameter or the return type for a function definition shall not be an incomplete (possibly `cv`-qualified) class type in the context of the function definition unless the function is deleted (11.4.3).

A typedef of function type may be used to declare a function but shall not be used to define a function (11.4). [Example:

```cpp
typedef void F();
F fv; // OK: equivalent to void fv();
F fv () {} // ill-formed
void fv () {} // OK: definition of fv
```

An identifier can optionally be provided as a parameter name; if present in a function definition (11.4), 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]

[Example: The declaration

```c
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

```c
typedef int IFUNC(int);
IFUNC* fpif(int);
```

or

```c
auto fpif(int)->int(*)(int);
```

A trailing-return-type is most useful for a type that would be more complicated to specify before the declarator-id:

```c
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```c
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 A declarator-id or abstract-declarator containing an ellipsis shall only be used in a parameter-declaration. Such a parameter-declaration is a parameter pack (17.6.3). When it is part of a parameter-declaration-clause, the parameter pack is a function parameter pack (17.6.3). [Note: Otherwise, the parameter-declaration is part of a template-parameter-list and the parameter pack is a template parameter pack; see 17.1. — end note] A function parameter pack is a pack expansion (17.6.3). [Example:

```c
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.¹⁰³

¹⁰³ 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).
11.3.6 Default arguments

If an initializer-clause is specified in a parameter-declaration this initializer-clause is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

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.

A default argument shall be specified only in the parameter-declaration-clause of a function declaration or lambda-declarator or in a template-parameter (17.1); in the latter case, the initializer-clause shall be an assignment-expression. A default argument shall not be specified for a 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.

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 or shall be a function parameter pack.

A default argument shall not be redefined by a later declaration (not even to the same value).

```
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
}
void n() {
  f(6);
  // OK, calls f(6, 7)
}
```

— 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 (11.6). 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 17.8.1. Example: In the following code, `g` will be called with the value `f(2)`:

```
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 Clause 14. —end note]

6 Except for member functions of class templates, the default arguments in a member function definition that appears outside of the class definition are added to the set of default arguments provided by the member function declaration in the class definition; the program is ill-formed if a default constructor (15.1), copy or move constructor, or copy or move assignment operator (15.8) 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]
```

7 [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]
```

8 [Note: The keyword this may not appear in a default argument of a member function; see 8.4.2. [Example:

```cpp
class A {
  void f(A* p = this) { }  
  // error
};
—end example]
```

9 A default argument is evaluated each time the function is called with no argument for the corresponding parameter. A parameter shall not appear as a potentially-evaluated expression in a default argument. Parameters of a function declared before a default argument are in scope and can hide namespace and class member names. [Example:

```cpp
int a;
int f(int a, int b = a);  
// error: parameter a used as default argument
typedef int I;
int g(float I, int b = I(2));  
// error: parameter I found
int h(int a, int b = sizeof(a));  
// OK, unevaluated operand
—end example]
```

A non-static member shall not appear in a default argument unless it appears as the id-expression of a class member access expression (8.5.1.5) or unless it is used to form a pointer to member (8.5.2.1). [Example: The declaration of X::mem1() in the following example is ill-formed because no object is supplied for the non-static member X::a used as an initializer.

```cpp
int b;
class X {
  int a;
  int mem1(int i = a);  
  // error: non-static member a used as default argument
  int mem2(int i = b);  
  // OK; use X::b
  static int b;
};
```

§ 11.3.6 191
The declaration of \( X::\text{mem2}() \) is meaningful, however, since no object is needed to access the static member \( X::b \). Classes, objects, and members are described in Clause 12. — end example] A default argument is not part of the type of a function. [ Example:

```c
void h() {
    int j = f(1);
    int k = f(); // OK, means f(0)
}
```

```c
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 (10.3.3), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also known at any point following the redeclaration where the using-declaration is in scope.

10 A virtual function call (13.3) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides. [ Example:

```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]

11.4 Function definitions [dcl.fct.def]

11.4.1 In general [dcl.fct.def.general]

1 Function definitions have the form

```
function-definition:
    attribute-specifier-seqopt declspecifier-seqopt declarator virt-specifier-seqopt function-body
    attribute-specifier-seqopt declspecifier-seqopt declarator requires-clause 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 (12.2).

2 In a function-definition, either void declarator ; or declarator ; shall be a well-formed function declaration as described in 11.3.5. A function shall be defined only in namespace or class scope.

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 declspecifier-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 15.1 and 15.6.

[Note: A `cv-qualifier-seq` affects the type of `this` in the body of a member function; see 11.3.2. — end note]

Unused parameters need not be named. For example,

```cpp
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` `__func__` is defined as if a definition of the form

```cpp
static const char __func__[] = "function-name";
```

had been provided, where `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.105

[Example:

```cpp
struct S {
  S() : s(__func__) { } // OK
  const char* s;
};
void f(const char* s = __func__); // error: __func__ is undeclared

—end example]

### 11.4.2 Explicitly-defaulted functions [dcl.fct.def.default]

1 A function definition whose `function-body` is of the form `= default ;` is called an `explicitly-defaulted` definition. A function that is explicitly defaulted shall

(1.1) — be a special member function or a comparison operator (8.5.8, 8.5.9, 8.5.10), and

(1.2) — not have default arguments.

2 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:

(2.1) — `T_1` and `T_2` may have differing `ref-qualifiers`; and

(2.2) — if `T_2` has a parameter of type `const C&`, the corresponding parameter of `T_1` may be of type `C&`.

If `T_1` differs from `T_2` in any other way, then:

(2.3) — 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;

(2.4) — otherwise, if `F` is explicitly defaulted on its first declaration, it is defined as deleted;

(2.5) — otherwise, the program is ill-formed.

3 An explicitly-defaulted function that is not defined as deleted may be declared `constexpr` only if it would have been implicitly declared as `constexpr`. If a function is explicitly defaulted on its first declaration, it is implicitly considered to be `constexpr` if the implicit declaration would be.

4 [Example:

```cpp
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; // deleted: exception specification does not match
private:
  int i;
  S(S&); // OK: private copy constructor
};
S::S(S&) = default; // OK: defines copy constructor

—end example]
```

105) 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.
Explicitly-defaulted functions and implicitly-declared functions are collectively called *defaulted* functions, and the implementation shall provide implicit definitions for them (15.1 15.4, 15.8), 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:**

```cpp
class 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
```

---

1.4.3 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 enforce non-default-initialization and non-integral initialization with

```cpp
struct onlydouble {
   onlydouble() = delete; // OK, but redundant
   onlydouble(std::intmax_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.

```cpp
struct sometype {
   void* operator new(std::size_t) = delete;
   void* operator new[] (std::size_t) = delete;
};
sometype* p = new sometype; // error, deleted class operator new
sometype* q = new sometype[3]; // error, deleted class operator new[]
```

---

**Example:** One can make a class uncopyable, i.e., move-only, by using deleted definitions of the copy constructor and copy assignment operator, and then providing defaulted definitions of the move constructor and move assignment operator.

```cpp
struct moveonly {
   moveonly() = default;
   moveonly(const moveonly&) = delete;
   moveonly(moveonly&&) = default;
   moveonly& operator=(const moveonly&) = delete;
   moveonly& operator=(moveonly&&) = default;
};
```
A deleted function is implicitly an inline function (10.1.6). [ 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 (6.6.4.4) shall not be defined as deleted. [ Example:]

```cpp
struct sometype {
    sometype();
};
sometype::sometype() = delete; // ill-formed; not first declaration
— end example ]
```

### 11.5 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), called structured bindings. Let \(cv\) denote the cv-qualifiers in the decl-specifier-seq. 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, \(e\) has type \(cv\ A\) 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

\[
\texttt{attribute-specifier-seq opt decl-specifier-seq ref-qualifier opt e} \text{ initializer ;}
\]

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 (8.2). — end note ]

If \(E\) is an array type with element type \(T\), the number of elements in the identifier-list shall be equal to the number of elements of \(E\). Each \(v_i\) is the name of an lvalue that refers to the element \(i\) of the array and whose type is \(T\); the referenced type is \(T\). [ Note: The top-level cv-qualifiers of \(T\) are \(cv\). — end note ] [ Example:]

```cpp
auto f() -> int(&)[2];
auto [ x, y ] = f(); // x and y refer to elements in a copy of the array return value
auto & [ xr, yr ] = f(); // xr and yr refer to elements in the array referred to by f’s return value
— end example ]
```

Otherwise, if the qualified-id \(\texttt{std::tuple_size<E>}\) names a complete type, the expression \(\texttt{std::tuple_size<E>::value}\) shall be a well-formed integral constant expression and the number of elements in the identifier-list shall be equal to the value of that expression. The unqualified-id \(\texttt{get}\) is looked up in the scope of \(E\) by class member access lookup (6.4.5), and if that finds at least one declaration, the initializer is \(e.\texttt{get<i>}()\). Otherwise, the initializer is \(\texttt{get<i>}(e)\), where \(\texttt{get}\) is looked up in the associated namespaces (6.4.2). In either case, \(\texttt{get<i>}\) 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 \(\texttt{std::tuple_element<i, E>::type}\), variables are introduced with unique names \(r_i\) of type “reference to \(T_i\)” initialized with the initializer (11.6.3), where the reference is an lvalue reference if the initializer is an lvalue and an rvalue reference otherwise. 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 public direct members of \(E\) or of the same unambiguous public base class of \(E\), \(E\) 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:]

```cpp
struct S { int x1 : 2; volatile double y1; };  
S f();
const auto [ x, y ] = f();
```

§ 11.5 195
The type of the id-expression x is "const int", the type of the id-expression y is "const volatile double".
— end example]

11.6 Initializers

The process of initialization described in this subclause applies to all initializations regardless of syntactic context, including the initialization of a function parameter (8.5.1.2), the initialization of a return value (9.6.3), or when an initializer follows a declarator.

\[
\text{initializer: brace-or-equal-initializer ( expression-list )}
\]

\[
\text{brace-or-equal-initializer: = initializer-clause braced-init-list}
\]

\[
\text{initializer-clause: assignment-expression braced-init-list}
\]

\[
\text{braced-init-list: \{ initializer-list , opt \}}
\]

\[
\text{\{ designated-initializer-list , opt \}}
\]

\[
\text{\{ \}}
\]

\[
\text{initializer-list: initializer-clause \ldots opt}
\]

\[
\text{initializer-list , initializer-clause \ldots opt}
\]

\[
\text{designated-initializer-list: designated-initializer-clause designated-initializer-list , designated-initializer-clause}
\]

\[
\text{designated-initializer-clause: designator brace-or-equal-initializer}
\]

\[
\text{designator: . identifier}
\]

\[
\text{expr-or-braced-init-list: expression braced-init-list}
\]

\[
\text{Note: The rules in this subclause apply even if the grammar permits only the brace-or-equal-initializer form of initializer in a given context. — end note}
\]

Except for objects declared with the constexpr specifier, for which see 10.1.5, 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:

\[
\text{int f(int);}
\]

\[
\text{int a = 2;}
\]

\[
\text{int b = f(a);}
\]

\[
\text{int c(b);}
\]

— end example]

\[
\text{Note: Default arguments are more restricted; see 11.3.6. — end note}
\]

[Note: The order of initialization of variables with static storage duration is described in 6.8.3 and 9.7. — end note]

A declaration of a block-scope variable with external or internal linkage that has an initializer is ill-formed.

To zero-initialize an object or reference of type T means:

\[
\text{— if T is a scalar type (6.7), the object is initialized to the value obtained by converting the integer literal 0 (zero) to T};^\text{106}
\]

\[
\text{— if T is a (possibly cv-qualified) non-union class type, its padding bits (6.7) are initialized to zero bits and each non-static data member, each non-virtual base class subobject, and, if the object is not a base class subobject, each virtual base class subobject is zero-initialized;}
\]

\text{106) As specified in 7.11, converting an integer literal whose value is 0 to a pointer type results in a null pointer value.}
(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:

(7.1) if \( T \) is a (possibly cv-qualified) class type (Clause 12), constructors are considered. The applicable constructors are enumerated (16.3.1.3), and the best one for the \( \text{initializer} () \) is chosen through overload resolution (16.3). The constructor thus selected is called, with an empty argument list, to initialize the object.

(7.2) if \( T \) is an array type, each element is default-initialized.

(7.3) otherwise, no initialization is performed.

A class type \( T \) is const-default-constructible if default-initialization of \( T \) would invoke a user-provided constructor of \( T \) (not inherited from a base class) or if

(7.4) each direct non-variant non-static data member \( M \) of \( T \) has a default member initializer or, if \( M \) is of class type \( X \) (or array thereof), \( X \) is const-default-constructible,

(7.5) if \( T \) is a union with at least one non-static data member, exactly one variant member has a default member initializer,

(7.6) if \( T \) is not a union, for each anonymous union member with at least one non-static data member (if any), exactly one non-static data member has a default member initializer, and

(7.7) each potentially constructed base class of \( T \) is const-default-constructible.

If a program calls for the default-initialization of an object of a const-qualified type \( T \), \( T \) shall be a const-default-constructible class type or array thereof.

8 To value-initialize an object of type \( T \) means:

(8.1) if \( T \) is a (possibly cv-qualified) class type (Clause 12) with either no default constructor (15.1) or a default constructor that is user-provided or deleted, then the object is default-initialized;

(8.2) 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;

(8.3) if \( T \) is an array type, then each element is value-initialized;

(8.4) otherwise, the object is zero-initialized.

9 A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

10 [Note: Every object of static storage duration is zero-initialized at program startup before any other initialization takes place. In some cases, additional initialization is done later. — end note]

11 An object whose initializer is an empty set of parentheses, i.e., (), shall be value-initialized.

[Note: Since () is not permitted by the syntax for initializer, \( X \ a() \); is not the declaration of an object of class \( X \), but the declaration of a function taking no argument and returning an \( X \). The form () is permitted in certain other initialization contexts (8.5.2.4, 8.5.1.3, 15.6.2). — end note]

12 If no initializer is specified for an object, the object is default-initialized. 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 (8.5.18). [Note: Objects with static or thread storage duration are zero-initialized, see 6.8.3.2. — end note] If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:

(12.1) If an indeterminate value of unsigned narrow character type (6.7.1) or std::byte type (21.2.1) is produced by the evaluation of:

(12.1.1) the second or third operand of a conditional expression (8.5.16),

(12.1.2) the right operand of a comma expression (8.5.19),

§ 11.6
— the operand of a cast or conversion (7.8, 8.5.1.3, 8.5.1.9, 8.5.3) to an unsigned narrow character type or `std::byte` type (21.2.1), or

— a discarded-value expression (8.2),

then the result of the operation is an indeterminate value.

— If an indeterminate value of unsigned narrow character type or `std::byte` type is produced by the evaluation of the right operand of a simple assignment operator (8.5.18) whose first operand is an lvalue of unsigned narrow 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 narrow character type is produced by the evaluation of the initialization expression when initializing an object of unsigned narrow character type, that object is initialized to an indeterminate value.

— If an indeterminate value of unsigned narrow 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]

13 An initializer for a static member is in the scope of the member's class. [Example:

```cpp
int a;
struct X {
    static int a;
    static int b;
};
int X::a = 1;
int X::b = a;  // X::b = X::a
```

—end example]

14 If the entity being initialized does not have class type, the *expression-list* in a parenthesized initializer shall be a single expression.

15 The initialization that occurs in the = form of a `brace-or-equal-initializer` or *condition* (9.4), as well as in argument passing, function return, throwing an exception (18.1), handling an exception (18.3), and aggregate member initialization (11.6.1), is called *copy-initialization*. [Note: Copy-initialization may invoke a move (15.8). — end note]

16 The initialization that occurs in the forms

```cpp
T x(a);
T x(a);
```

as well as in *new* expressions (8.5.2.4), *static_cast* expressions (8.5.1.9), functional notation type conversions (8.5.1.3), *mem-initializers* (15.6.2), and the *braced-init-list* form of a *condition* is called *direct-initialization*.

17 The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. If the initializer is not a single (possibly parenthesized) expression, the source type is not defined.

— If the initializer is a (non-parenthesized) `brace-init-list` or is = `brace-init-list`, the object or reference is list-initialized (11.6.4).

— If the destination type is a reference type, see 11.6.3.

— If the destination type is an array of characters, an array of `char16_t`, an array of `char32_t`, or an array of `wchar_t`, and the initializer is a string literal, see 11.6.2.
If the initializer is \( () \), the object is value-initialized.

Otherwise, if the destination type is an array, the program is ill-formed.

If the destination type is a (possibly \( \text{cv} \)-qualified) class type:

- If the initializer expression is a prvalue and the \( \text{cv} \)-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: \( \text{T x} = \text{T(T())} \); calls the \( \text{T} \) default constructor to initialize \( x \). —end example]

- Otherwise, if the initialization is direct-initialization, or if it is copy-initialization where the \( \text{cv} \)-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 (16.3.1.3), and the best one is chosen through overload resolution (16.3). The constructor so selected is called to initialize the object, with the initializer expression or expression-list as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.

- Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversion sequences 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 16.3.1.4, and the best one is chosen through overload resolution (16.3). 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 \( \text{cv} \)-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 \( \text{cv} \)-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated (16.3.1.5), and the best one is chosen through overload resolution (16.3). 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. 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 \( \text{cv1 T} \) can initialize an object of type \( \text{cv2 T} \) independently of the \( \text{cv} \)-qualifiers \( \text{cv1} \) and \( \text{cv2} \).]

An \text{initializer-clause} followed by an ellipsis is a pack expansion (17.6.3).

If the initializer is a parenthesized expression-list, the expressions are evaluated in the order specified for function calls (8.5.1.2).

The same \text{identifier} shall not appear in multiple \text{designators} of a \text{designated-initializer-list}.

An object whose initialization has completed is deemed to be constructed, even if 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 (11.6.1) or by an inherited constructor (15.6.3). —end note]

A declaration that specifies the initialization of a variable, whether from an explicit initializer or by default-initialization, is called the initial\text{izing 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 (12.2.3.2) might be the declaration within the class definition and not the definition at namespace scope. —end note]

\textbf{11.6.1 Aggregates} [dcl.init.aggr]

An \text{aggregate} is an array or a class (Clause 12) with

- no user-provided, \text{explicit}, or inherited constructors (15.1),
- no private or protected non-static data members (Clause 14),
— no virtual functions (13.3), and
— no virtual, private, or protected base classes (13.1).

[Note: Aggregate initialization does not allow accessing protected and private base class’ members or constructors. —end note]

2 The elements of an aggregate are:

— for an array, the array elements in increasing subscript order, or
— for a class, the direct base classes in declaration order, followed by the direct non-static data members (12.2) that are not members of an anonymous union, in declaration order.

3 When an aggregate is initialized by an initializer list as specified in 11.6.4, 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:

— 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.
— 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.
— Otherwise, the initializer list must be \( \{ \} \), and there are no explicitly initialized elements.

For each explicitly initialized element:

— 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.
— Otherwise, the element is copy-initialized from the corresponding initializer-clause or 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 (11.6.4) 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:

```
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.
```

```
struct base1 { int b1, b2 = 42; };
struct base2 {
  base2() {
    b3 = 42;
  }
  int b3;
};
struct derived : base1, base2 {
  int d;
};
```

```
derived d1{1, 2}, { }, 4;
derived d2{0}, { }, 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]
```

5 For a non-union aggregate, each element that is not an explicitly initialized element is initialized as follows:
If the element has a default member initializer (12.2), the element is initialized from that initializer.

Otherwise, if the element is not a reference, the element is copy-initialized from an empty initializer list (11.6.4).

Otherwise, the program is ill-formed.

If the aggregate is a union and the initializer list is empty, then

— if any variant member has a default member initializer, that member is initialized from its default member initializer;

— otherwise, the first member of the union (if any) is copy-initialized from an empty initializer list.

[Example:

```cpp
struct S { int a; const char* b; int c; int d = b[a]; };  
S ss = { 1, "asdf" };  
```

initializes `ss.a` with 1, `ss.b` with "asdf", `ss.c` with the value of an expression of the form `int{}` (that is, 0), and `ss.d` with the value of `ss.b[ss.a]` (that is, 's'), and in

```cpp
struct X { int i, j, k = 42; };  
X a[] = { 1, 2, 3, 4, 5, 6 };  
X b[2] = { { 1, 2, 3 }, { 4, 5, 6 } };  
```

`a` and `b` have the same value

```cpp
struct A {
    string a;
    int b = 42;
    int c = -1;
};  
A{.c=21} has the following steps:

— Initialize `a` with `{}`

— Initialize `b` with = 42

— Initialize `c` with = 21

—end example]

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 11.6.

An array of unknown bound initialized with a brace-enclosed `initializer-list` containing `n` `initializer- clauses`, where `n` shall be greater than zero, is defined as having `n` elements (11.3.4). [Example:

```cpp
int x[] = { 1, 3, 5 };  
```

declares and initializes `x` as a one-dimensional array that has three elements since no size was specified and there are three initializers. —end example] An empty initializer list `{}` shall not be used as the `initializer-clause` for an array of unknown bound.\[107\] [Note: A default member initializer does not determine the bound for a member array of unknown bound. Since the default member initializer is ignored if a suitable `mem-initializer` is present (15.6.2), the default member initializer is not considered to initialize the array of unknown bound. [Example:

```cpp
struct S {
    int y[] = { 0 };  // error: non-static data member of incomplete type
};  
```

—end example] —end note]

[Note: Static data members, non-static data members of anonymous union members, and unnamed bit-fields are not considered elements of the aggregate. [Example:

```cpp
struct A {
    int i;
};
```

\[107\] The syntax provides for empty `initializer-lists`, but nonetheless C++ does not have zero length arrays.
static int s;
int j;
int :17;
int k;
} a = { 1, 2, 3 };

Here, the second initializer 2 initializes \( a.j \) and not the static data member \( A::s \), and the third initializer 3 initializes \( a.k \) and not the unnamed bit-field before it. —end example —end note

10 An initializer-list is ill-formed if the number of initializer-clauses exceeds the number of elements of the aggregate. [Example:

\[
\text{char cv}[4] = \{ 'a', 's', 'd', 'f', 0 \}; \quad // \text{error}
\]

is ill-formed. —end example]

11 If a reference member is initialized from its default member initializer and a potentially-evaluated subexpression thereof is an aggregate initialization that would use that default member initializer, the program is ill-formed. [Example:

\[
\text{struct A;}
\text{extern A a;}
\text{struct A \{}
\text{\quad const A& a1 \{ A(a,a) \}; \quad // OK}
\text{\quad const A& a2 \{ A() \}; \quad // error}
\text{\};}
\text{A a(a,a);} \quad // \text{OK}
\]

—end example]

12 If an aggregate class \( C \) contains a subaggregate element \( e \) with no elements, the initializer-clause for \( e \) shall not be omitted from an initializer-list for an object of type \( C \) unless the initializer-clauses for all elements of \( C \) following \( e \) are also omitted. [Example:

\[
\text{struct S \{ \} s;}
\text{struct A \{}
\text{\quad S s1;}
\text{\quad int i1;}
\text{\quad S s2;}
\text{\quad int i2;}
\text{\quad S s3;}
\text{\quad int i3;}
\text{\} a = \{}
\text{\quad \{ \}, \quad // \text{Required initialization}
\text{\quad 0, \quad \quad \quad // \text{Required initialization}
\text{\quad s, \quad \quad \quad // \text{Initialization not required for } A::s3 \text{ because } A::i3 \text{ is also not initialized}
\text{\quad 0 \};}
\]

—end example]

13 When initializing a multi-dimensional array, the initializer-clauses initialize the elements with the last (rightmost) index of the array varying the fastest (11.3.4). [Example:

\[
\text{int x[2][2] = \{ 3, 1, 4, 2 \};}
\]

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,

\[
\text{float y[4][3] = \{}
\text{\quad \{ 1 \}, \{ 2 \}, \{ 3 \}, \{ 4 \}
\text{\};}
\]

initializes the first column of \( y \) (regarded as a two-dimensional array) and leaves the rest zero. —end example]

14 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:
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,

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 (Clause 7) 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:
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-provided constructor (15.1). Initialization of these aggregate objects is described in 15.6.1. —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 9.7. —end note]

When a union is initialized with an initializer list, there shall not be more than one explicitly initialized element. [Example:
union u { int a; const char* b; }
    u a = { 1 };
    u b = a;
    u c = 1;   // error
    u d = { 0, "asdf" };   // error
    u e = { "asdf" };   // error
    u f = { .b = "asdf" };   // error
    u g = { .a = 1, .b = "asdf" };   // error
 —end example]

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]

11.6.2 Character arrays [dcl.init.string]

An array of narrow character type (6.7.1), char16_t array, char32_t array, or wchar_t array can be initialized by a narrow string literal, char16_t string literal, char32_t 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:
char msg[] = "Syntax error on line %s\n";
shows a character array whose members are initialized with a string-literal. Note that because ‘\n’ is a single character and because a trailing ‘\0’ is appended, sizeof(msg) is 25. —end example 2

There shall not be more initializers than there are array elements. [Example:
char cv[4] = "asdf"; // error
is ill-formed since there is no space for the implied trailing ‘\0’. —end example 2]

If there are fewer initializers than there are array elements, each element not explicitly initialized shall be zero-initialized (11.6).

11.6.3 References [dcl.init.ref]
1 A variable whose declared type is “reference to type T” (11.3.2) shall be initialized. [Example:
int g(int) noexcept;
void f() {
  int i;
  int& r = i; // r refers to i
  r = 1; // the value of i becomes 1
  int* p = &r; // p points to i
  int& rr = r; // rr refers to what r refers to, that is, to i
  int (&rg)(int) = g; // rg refers to the function g
  rg(i); // calls function g
  int a[3];
  int (&ra)[3] = a; // ra refers to the array a
  ra[1] = i; // modifies a[1]
}
—end example 2]

2 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 (8.5.18). —end note] Argument passing (8.5.1.2) and function value return (9.6.3) are initializations.

3 The initializer can be omitted for a reference only in a parameter declaration (11.3.5), in the declaration of a function return type, in the declaration of a class member within its class definition (12.2), and where the extern specifier is explicitly used. [Example:
int& r1; // error: initializer missing
extern int& r2; // OK
—end example 3]

4 Given types “cv1 T1” and “cv2 T2”, “cv1 T1” is reference-related to “cv2 T2” if T1 is the same type as T2, or T1 is a base class of T2. “cv1 T1” is reference-compatible with “cv2 T2” if
 (4.1) T1 is reference-related to T2, or
 (4.2) T2 is “noexcept function” and T1 is “function”, where the function types are otherwise the same, and cv1 is the same cv-qualification as, or greater cv-qualification than, cv2. In all cases where the reference-related or reference-compatible relationship of two types is used to establish the validity of a reference binding, and T1 is a base class of T2, a program that necessitates such a binding is ill-formed if T1 is an inaccessible (Clause 14) or ambiguous (13.2) base class of T2.

5 A reference to type “cv1 T1” is initialized by an expression of type “cv2 T2” as follows:
 (5.1) If the reference is an lvalue reference and the initializer expression
 (5.1.1) is an lvalue (but is not a bit-field), and “cv1 T1” is reference-compatible with “cv2 T2”, or
 (5.1.2) has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be converted to an lvalue of type “cv3 T3”, where “cv1 T1” is reference-compatible with “cv3 T3”108 (this conversion is selected by enumerating the applicable conversion functions (16.3.1.6) and choosing the best one through overload resolution (16.3)),

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

108 This requires a conversion function (15.3.2) returning a reference type.
object). [Note: The usual lvalue-to-rvalue (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. — end note]

[Example:

double d = 2.0;
double& rd = d; // rd refers to d
const double& rcd = d; // rcd refers to d

struct A {
};
struct B : A { operator int&(); } b;
A& ra = b; // ra refers to A subobject in b
const A& rca = b; // rca refers to A subobject in b
int& ir = B(); // ir refers to the result of B::operator int

— end example]

— Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., cv1 shall be const), or the reference shall be an rvalue reference. [Example:

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]

— Otherwise:

— If the initializer expression

— is an rvalue (but not a bit-field) or function lvalue and “cv1 T1” is reference-compatible with “cv2 T2”, or

— has a class type (i.e., T2 is a class type), where T1 is not reference-related to T2, and can be converted to an rvalue or function lvalue of type “cv3 T3”, where “cv1 T1” is reference-compatible with “cv3 T3” (see 16.3.1.6),

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” (7.5) and the temporary materialization conversion (7.4) is applied. In any case, the reference is bound to the resulting glvalue (or to an appropriate base class subobject).

[Example:

struct A {
};
struct B : A { } b;
extern B f();
const A& rca2 = f(); // bound to the A subobject of the B rvalue.
A& rra = f(); // same as above
struct X {
    operator B();
    operator int&();
} x;
const 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]

— Otherwise:

— 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 (11.6, 16.3.1.4, 16.3.1.5); 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.

— 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 \( T_1 \) is reference-related to \( T_2 \):

- \( cv_1 \) shall be the same cv-qualification as, or greater cv-qualification than, \( cv_2 \); and
- if the reference is an rvalue reference, the initializer expression shall not be an lvalue.

[Example:

```cpp
class Banana { };  // T1
class Enigma { operator const Banana(); };  // T2
class Alaska { operator Banana&(); };  // T2

void enigmatic() {
    typedef const Banana ConstBanana;
    Banana &&banana1 = ConstBanana();  // ill-formed
    Banana &&banana2 = Enigma();  // ill-formed
    Banana &&banana3 = Alaska();  // ill-formed
}
```

```cpp
class A { operator volatile int&(); };

const int& r2 = cvi;  // error: cv-qualifier dropped
```

```cpp
double d2 = 1.0;  // d2 refers to temporary with value 2.0
```

```cpp
double&& rrd = 2;  // rrd refers to temporary with value 2.0
```

```cpp
const volatile int cvi = 1;  // error: cv-qualifier dropped
```

```cpp
struct A { operator volatile int&(); } a;
```

```cpp
const int& r3 = a;  // error: cv-qualifier dropped  // from result of conversion function
```

```cpp
double d2 = 1.0;
```

```cpp
const volatile int cvi = 1;
```

```cpp
const int& r2 = cvi;  // error: initializer is lvalue of related type
```

```cpp
struct A { operator int&(); } a;
```

```cpp
const int& r3 = a;  // error: cv-qualifier dropped  // from result of conversion function
```

```cpp
double d2 = 1.0;
```

```cpp
double&& rrd2 = d2;  // error: initializer is lvalue of related type
```

```cpp
struct X { operator int&(); };
```

```cpp
int&& rri2 = X();  // error: result of conversion function is lvalue of related type
```

```cpp
int i3 = 2;
```

```cpp
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 underlying type of the reference), the reference is said to bind directly to the initializer expression.

[Note: 15.2 describes the lifetime of temporaries bound to references. — end note]

### 11.6.4 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

- as the initializer in a variable definition (11.6)
- as the initializer in a new-expression (8.5.2.4)
- in a return statement (9.6.3)
- as a for-range-initializer (9.5)
- as a function argument (8.5.1.2)
- as a subscript (8.5.1.1)
- as an argument to a constructor invocation (11.6, 8.5.1.3)
- as an initializer for a non-static data member (12.2)
- in a mem-initializer (15.6.2)
- on the right-hand side of an assignment (8.5.18)

[Example:

```cpp
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
```
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 (11.3.6). [Note: Initializer-list constructors are favored over other constructors in list-initialization (16.3.1.7). 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 (17.9.2.1). — 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 (10.1.7.4) — the program is ill-formed.

List-initialization of an object or reference of type T is defined as follows:

1. If the braced-init-list contains a designated-initializer-list, T shall be an aggregate class. The ordered identifiers in the designators of the designated-initializer-list shall form a subsequence of the ordered identifiers in the direct non-static data members of T. Aggregate initialization is performed (11.6.1).
   
   ```
   struct A { int x; int y; int z; };        // error: designator order does not match declaration order
   A a(.y = 2, .x = 1);                   // OK, b.y initialized to 0
   A b(.x = 1, .z = 2);                  // OK
   ```
   — end example

2. If T is an aggregate class and the initializer list has a single element of type `cv U`, where U is T or a class derived from T, the object is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization).

3. Otherwise, if T is a character array and the initializer list has a single element that is an appropriately-typed string literal (11.6.2), initialization is performed as described in that subclause.

4. Otherwise, if T is an aggregate, aggregate initialization is performed (11.6.1).
   ```
   double ad[] = { 1, 2.0 };        // OK
   int ai[] = { 1, 2.0 };           // error: narrowing
   ```
   — end example

5. Otherwise, if the initializer list has no elements and T is a class type with a default constructor, the object is value-initialized.

6. Otherwise, if T is a specialization of `std::initializer_list<E>`, the object is constructed as described below.

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 (16.3.1.7). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

   ```
   struct S {
   S(std::initializer_list<double>);   // #1
   S(std::initializer_list<int>);      // #2
   S();                                // #3
   ```
// ...
S s1 = { 1.0, 2.0, 3.0 };  // invoke #1
S s2 = { 1, 2, 3 };       // invoke #2
S s3 = { };              // invoke #3

— end example

[Example:
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);    // #1
  S();                        // #2
  // ...
};
S s1 = { 1, 2, 3.0 };       // OK: invoke #1
S s2 { 1.0, 2, 3 };         // error: narrowing
S s3 { };                   // OK: invoke #2

— end example

(3.8) — Otherwise, if T is an enumeration with a fixed underlying type (10.2), the initializer-list has a single element v, and the initialization is direct-list-initialization, the object is initialized with the value T(v) (8.5.1.3); if a narrowing conversion is required to convert v to the underlying type of T, the program is ill-formed. [Example:
enum byte : unsigned char { }; byte b { 42 };        // OK byte c { 42 };        // error byte d = byte{ 42 };       // OK; same value as b byte e { -1 };         // error

struct A { byte b; }; A a1 = { { 42 } };        // error A a2 = { byte{ 42 } };       // OK

void f(byte); f({ 42 });                      // error

enum class Handle : uint32_t { Invalid = 0 }; Handle h { 42 };       // OK

— end example]

(3.9) — Otherwise, if the initializer list has a single element of type E and either T is not a reference type or its referenced type is reference-related to E, the object or reference is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization); if a narrowing conversion (see below) is required to convert the element to T, the program is ill-formed. [Example:
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 or direct-list-initialization, depending on the kind of initialization for the reference. 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 an lvalue reference to a non-const type. — end note]

§ 11.6.4
Example:

```cpp
struct S {
    S(std::initializer_list<double>); // #1
    S(const std::string&); // #2
    // ...
};

const S& r1 = { 1, 2, 3.0 }; // OK: invoke #1
const S& r2 { "Spinach" }; // OK: invoke #2
S& r3 = { 1, 2, 3 }; // error: initializer is not an lvalue
const int& i1 = { 1 }; // OK
const int& i2 = { 1.1 }; // error: narrowing
const int (&iar)[2] = { 1, 2 }; // OK: iar is bound to temporary array

— end example
```

— Otherwise, if the initializer list has no elements, the object is value-initialized.

Example:

```cpp
int** pp {};
// initialized to null pointer
— end example
```

— Otherwise, the program is ill-formed.

Example:

```cpp
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 (17.6.3), 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 (7.4) 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 (Clause 14) 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 }
```

The initialization will be implemented in a way roughly equivalent to this:

```cpp
const double __a[3] = {double{1}, double{2}, double{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 (15.2), 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:

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 (15.6.2). — 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

— from a floating-point type to an integer type, or

— from long double to double or float, or from double to float, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly), or

— from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

— from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression whose value after integral promotions will fit into the target type.

[Note: As indicated above, such conversions are not allowed at the top level in list-initializations. — end note] [Example:

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);
char c3(y); // error: might narrow
char c4(z); // OK: no narrowing needed

unsigned char uc1 = {5}; // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrowing

unsigned int ui1 = {-1}; // error: narrowing

signed int si1 =
  { (unsigned int)-1 }; // error: narrowing
int ii1 = {2.0}; // error: narrowing
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]
12 Classes

A class is a type. Its name becomes a class-name (12.1) within its scope.

```
class-name:
  identifier
  simple-template-id
```

Class-specifiers and elaborated-type-specifiers (10.1.7.3) are used to make class-names. 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-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.

```
struct A;
struct A final {}; // OK: definition of struct A,
                    // not value-initialization of variable final
```

3 If a class is marked with the class-virt-specifier final and it appears as a class-or-decltype in a base-clause (Clause 13), 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 X {
  struct C { constexpr operator int() { return 5; } };
  struct B final : C{}; // OK: definition of nested class B,
                        // not declaration of a bit-field member final
};
```

—end example]

4 Complete objects and member subobjects of class type shall have nonzero size. [Note: Class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying or moving has been restricted; see 15.8). Other plausible operators, such as equality comparison, can be defined by the user; see 16.5. —end note]

5 A union is a class defined with the class-key union; it holds at most one data member at a time (12.3). [Note: Aggregates of class type are described in 11.6.1. —end note]

6 A trivially copyable class is a class:

---

109) Base class subobjects are not so constrained.
(6.1) — where each copy constructor, move constructor, copy assignment operator, and move assignment operator (15.8, 16.5.3) is either deleted or trivial,
(6.2) — that has at least one non-deleted copy constructor, move constructor, copy assignment operator, or move assignment operator, and
(6.3) — that has a trivial, non-deleted destructor (15.4).

A trivial class is a class that is trivially copyable and has one or more default constructors (15.1), 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]

7 A class $S$ is a standard-layout class if it:
(7.1) — has no non-static data members of type non-standard-layout class (or array of such types) or reference,
(7.2) — has no virtual functions (13.3) and no virtual base classes (13.1),
(7.3) — has the same access control (Clause 14) for all non-static data members,
(7.4) — has no non-standard-layout base classes,
(7.5) — has at most one base class subobject of any given type,
(7.6) — has all non-static data members and bit-fields in the class and its base classes first declared in the same class, and
(7.7) — has no element of the set $M(S)$ of types (defined below) as a base class.\(^{110}\)

$M(X)$ is defined as follows:
(7.8) — If $X$ is a non-union class type with no (possibly inherited (Clause 13)) non-static data members, the set $M(X)$ is empty.
(7.9) — If $X$ is a non-union class type whose first non-static data member has type $X_0$ (where said member may be an anonymous union), the set $M(X)$ consists of $X_0$ and the elements of $M(X_0)$.
(7.10) — If $X$ is a union type, the set $M(X)$ is the union of all $M(U_i)$ and the set containing all $U_i$, where each $U_i$ is the type of the $i$th non-static data member of $X$.
(7.11) — If $X$ is an array type with element type $X_e$, the set $M(X)$ consists of $X_e$ and the elements of $M(X_e)$.
(7.12) — If $X$ is a non-class, non-array type, the set $M(X)$ is empty.

[Note: $M(X)$ is the set of the types of all non-base-class subobjects that are guaranteed in a standard-layout class to be at a zero offset in $X$. — end note]

[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]

8 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 12.2. — end note]

[Example:
```
struct N {
  int i; // neither trivial nor standard-layout
  int j;
```

110] 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 (8.5.10).
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 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 (10.3.1) 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.

12.1 Class names [class.name]

1 A class definition introduces a new type. [Example:

    struct X { int a; };
    struct Y { int a; };
    X a1;
    Y a2;
    int a3;

defines three variables of three different types. This implies that

    a1 = a2;  // error: Y assigned to X
    a1 = a3;  // error: int assigned to X

despite that

    int f(X);
    int f(Y);

declare an overloaded (Clause 16) function f() and not simply a single function f() twice. For the same reason,

    struct S { int a; };
    struct S { int a; };  // error, double definition

is ill-formed because it defines S twice. —end example]

2 A class declaration introduces the class name into the scope where it is declared and hides any class, variable, function, or other declaration of that name in an enclosing scope (6.3). 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:

    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 14.3, operator functions in 16.5. —end example] —end note]

3 [Note: An elaborated-type-specifier (10.1.7.3) 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 typedef-name (10.1.3) that names a class type, or a cv-qualified version thereof, is also a class-name. If a typedef-name that names a cv-qualified class type is used where a class-name is required, the cv-qualifiers are ignored. A typedef-name shall not be used as the identifier in a class-head.
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 (12.3.1) and direct members thereof. Members of a class are data members, member functions (12.2.1), nested types, enumerators, and member templates (17.6.2) 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

(2.1) — a friend declaration (14.3),
(2.2) — a static_assert-declaration,
(2.3) — a using-declaration (10.3.3), or
(2.4) — an empty-declaration.

For any other member-declaration, each declared entity that is not an unnamed bit-field (12.2.4) 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 (12.1, 12.2.5) and enumerations (10.2) declared in the class and arbitrary types declared as members by use of a typedef declaration (10.1.3) or alias-declaration. The enumerators of an unscoped enumeration (10.2) 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 12.2.3) (a static data member (12.2.3.2) or static member function (12.2.3.1), 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 (12.2.2), 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

(5.1) — a nested class or member class template can be declared and then later defined, and
(5.2) — an enumeration can be introduced with an opaque-enum-declaration and later redeclared with an enum-specifier.
A class is considered a completely-defined object type (6.7) (or complete type) at the closing \} of the class-specifier. Within the class member-specification, the class is regarded as complete within function bodies, default arguments, noexcept-specifiers, and default member initializers (including such things in nested classes). Otherwise it is regarded as incomplete within its own class member-specification.

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]

A brace-or-equal-initializer shall appear only in the declaration of a data member. (For static data members, see 12.2.3.2; for non-static data members, see 15.6.2 and 11.6.1). 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.

A member shall not be declared with the extern storage-class-specifier. Within a class definition, a member shall not be declared with the thread_local storage-class-specifier unless also declared static.

The decl-specifier-seq may be omitted in constructor, destructor, and conversion function declarations only; when declaring another kind of member the decl-specifier-seq shall contain a type-specifier that is not a cv-qualifier. The member-declarator-list can be omitted only after a class-specifier or an enum-specifier or in a friend declaration (14.3). A pure-specifier shall be used only in the declaration of a virtual function (13.3) that is not a friend declaration.

A simple example of a class definition is

```c
struct tnode {
    char tword[20];
};
```

\[ Note: \] A single name can denote several member functions provided their types are sufficiently different (Clause 16). —end note

\[ Note: \] The type of a non-static member function is an ordinary function type, and the type of a non-static data member is an ordinary object type. There are no special member function types or data member types. —end note

\[ Example: \] A simple example of a class definition is

```c
struct tnode {
    char tword[20];
};
```
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]

18 Non-static data members of a (non-union) class with the same access control (Clause 14) 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 (Clause 14). 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 (13.3) and virtual base classes (13.1).

19 If T is the name of a class, then each of the following shall have a name different from T:

(19.1) — every static data member of class T;
(19.2) — every member function of class T [Note: This restriction does not apply to constructors, which do not
have names (15.1) — end note] ;
(19.3) — every member of class T that is itself a type;
(19.4) — every member template of class T;
(19.5) — every enumerator of every member of class T that is an unscoped enumerated type; and
(19.6) — every member of every anonymous union that is a member of class T.

20 In addition, if class T has a user-declared constructor (15.1), every non-static data member of class T shall
have a name different from T.

21 The common initial sequence of two standard-layout struct (Clause 12) 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 and either neither entity is a bit-field
or both are bit-fields with the same width. [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]

22 Two standard-layout struct (Clause 12) types are layout-compatible classes if their common initial sequence
comprises all members and bit-fields of both classes (6.7).

23 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).

24 In a standard-layout union with an active member (12.3) of struct type T1, it is permitted to read a non-static
data member m of another union member of struct type T2 provided m is part of the common initial sequence
of T1 and T2; the behavior is as if the corresponding member of T1 were nominated. [Example:

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
}
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. Otherwise, its address is the same as the address of its first base class subobject (if any). [Note: There might therefore be unnamed padding within a standard-layout struct object, 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, 8.5.1.9). — end note]

12.2.1 Member functions

A member function may be defined (11.4) in its class definition, in which case it is an inline member function (10.1.6), 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 (17.8) appearing outside of the class definition, a member function shall not be redeclared.

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 in namespace scope have the linkage of that class. Member functions of a local class (12.4) have no linkage. See 6.5. — end 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 10.1.6. — end note]

If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the :: operator. [Note: A name used in a member function definition (that is, in the parameter-declaration-clause including the default arguments (11.3.6) 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);
    void X::f(T t = count) {}  // equivalent to: void f(T);
}
```

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]

A static local variable or local type in a member function always refers to the same entity, whether or not the member function is inline. — end note]

Previously declared member functions may be mentioned in friend declarations.

Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

[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 11.3.5. For example,

```c
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 17.3. — end note]

§ 12.2.1
12.2.2 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 (Clause 13) from its class type, using the class member access syntax (8.5.1.5, 16.3.1.1). A non-static member function may also be called directly using the function call syntax (8.5.1.2, 16.3.1.1) 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 (8.4) that is not part of a class member access syntax (8.5.1.5) and not used to form a pointer to member (8.5.2.1) is used in a member of class \(X\) in a context where \(this\) can be used (8.4.2), 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 (8.5.1.5) using (*this) (12.2.2.1) 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 \(unqualified-id\) (8.4) 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 \(qualified-id\) (8.4) in which the \(nested-name-specifier\) names the class of the member function. These transformations do not apply in the template definition context (17.7.2.1). [Example:

```c
struct tnode {
  char tword[20];
  int count;
  tnode* left;
  tnode* right;
  void set(const char*, tnode* l, tnode* r);
};
void tnode::set(const char* s, tnode* l, tnode* r) {
  count = strlen(s)+1;
  if (sizeof(tword)<=count)
    perror("tnode string too long");
  strcpy(tword,s);
  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. —end example]

A non-static member function may be declared \(const\), \(volatile\), or \(const\) \(volatile\). These \(cv\)-qualifiers affect the type of the \(this\) pointer (12.2.2.1). They also affect the function type (11.3.5) of the member function; a member function declared \(const\) is a \(const\) member function, a member function declared \(volatile\) is a \(volatile\) member function and a member function declared \(const\) \(volatile\) is a \(const\) \(volatile\) member function. [Example:

```c
struct X {
  void g() const;
  void h() const volatile;
};
```
\(X::g\) is a \(const\) member function and \(X::h\) is a \(const\) \(volatile\) member function. —end example]

A non-static member function may be declared with a \(ref\)-qualifier (11.3.5); see 16.3.1.

A non-static member function may be declared \(virtual\) (13.3) or \(pure virtual\) (13.4).
12.2.2.1 The this pointer

1 In the body of a non-static (12.2.1) member function, the keyword this is a prvalue expression whose value is the address of the object for which the function is called. The type of this in a member function of a class X is X*. If the member function is declared const, the type of this is const X*, if the member function is declared volatile, the type of this is volatile X*, and if the member function is declared const volatile, the type of this is const volatile X*. [Note: Thus in a const member function, the object for which the function is called is accessed through a const access path. —end note] [Example:

```c
struct s {
  int a;
  int f() const;
  int g() { return a++; }
  int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The a++ in the body of s::h is ill-formed because it tries to modify (a part of) the object for which s::h() is called. This is not allowed in a const member function because this is a pointer to const; that is, *this has const type. —end example]

2 Similarly, volatile semantics (10.1.7.1) apply in volatile member functions when accessing the object and its non-static data members.

3 A cv-qualified member function can be called on an object-expression (8.5.1.5) only if the object-expression is as cv-qualified or less-cv-qualified than the member function. [Example:

```c
void k(s& x, const s& y) {
  x.f();
  x.g();
  y.f();
  y.g(); // error
}
```

The call y.g() is ill-formed because y is const and s::g() is a non-const member function, that is, s::g() is less-qualified than the object-expression y. —end example]

4 Constructors (15.1) and destructors (15.4) shall not be declared const, volatile or const volatile. [Note: However, these functions can be invoked to create and destroy objects with cv-qualified types, see 15.1 and 15.4. —end note]

12.2.3 Static members

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 (8.5.1.5) 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:

```c
struct process {
  static void reschedule();
};

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 (Clause 13) 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:

```c
int g();
struct X {
  static int g();
};
```
3 If an unqualified-id (8.4) 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 8.4 for restrictions on the use of non-static data members and non-static member functions. —end note]

4 Static members obey the usual class member access rules (Clause 14). 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]

12.2.3.1 Static member functions [class.static.mfct]

1 [Note: The rules described in 12.2.1 apply to static member functions. —end note]

2 [Note: A static member function does not have a this pointer (12.2.2.1). —end note] A static member function shall not be virtual. There shall not be a static and a non-static member function with the same name and the same parameter types (16.1). A static member function shall not be declared const, volatile, or const volatile.

12.2.3.2 Static data members [class.static.data]

1 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.

2 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]

3 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 (8.6). 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.1). Declarations of other static data members shall not specify a brace-or-equal-initializer.
4 [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.

5 [Note: Static data members of a class in namespace scope have the linkage of that class (6.5). A local class cannot have static data members (12.4). —end note]

6 Static data members are initialized and destroyed exactly like non-local variables (6.8.3.2, 6.8.3.3, 6.8.3.4).

7 A static data member shall not be mutable (10.1.1).

12.2.4 Bit-fields

A member-declarator of the form

\[\text{identifier}_{\text{opt}} \text{ attribute-specifier-seq}_{\text{opt}} : \text{constant-expression brace-or-equal-initializer}_{\text{opt}}\]

specifies a bit-field; its length is set off from the bit-field name by a colon. The optional attribute-specifier-seq appertains to the entity being declared. The bit-field attribute 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. The value of the integral constant expression may be larger than the number of bits in the object representation (6.7) of the bit-field’s type; in such cases 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. [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 value of the constant-expression be equal to zero.

A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (6.7.1). A bool value can successfully be stored in a bit-field of any nonzero size. 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 (11.6.3). [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 11.6.3. —end note]

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 shall compare equal. If the value of an enumerator is stored into a bit-field of the same enumeration type and the number of bits in the bit-field is large enough to hold all the values of that enumeration type (10.2), the original enumerator value and the value of the bit-field shall compare equal. [Example:

```c
enum BOOL { FALSE=0, TRUE=1 };
struct A {
    BOOL b:1;
};
A a;
void f() {
    a.b = TRUE;
    if (a.b == TRUE) // yields true
        {/* ... */}
}
—end example]

12.2.5 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 8.4 for restrictions on the use of non-static data members and non-static member functions. —end note]

[Example:

```c
int x;
```
int y;

struct enclose {
    int x;
    static int s;
}

struct inner {
    void f(int i) {
        int a = sizeof(x);  // OK: operand of sizeof is an unevaluated operand
        x = i;  // error: assign to enclose::x
        s = i;  // OK: assign to enclose::s
        ::x = i;  // OK: assign to global x
        y = i;  // OK: assign to global y
    }
    void g(enclose* p, int i) {
        p->x = i;  // OK: assign to enclose::x
    }
};

inner* p = 0;  // error: inner not in scope

— end example

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
```

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
```

Like a member function, a friend function (14.3) 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 (12.2.3), but it has no special access rights to members of an enclosing class.

### 12.2.6 Nested type names

Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [Example:

```c
struct X {
    typedef int I;
    class Y { /* ... */ };  // error
    I a;
};

I b;  // error
Y c;  // error
```

§ 12.2.6
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 (12.2), 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 12.2. — 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 struct. [Note: A union object and its non-static data members are pointer-interconvertible (6.7.2, 8.5.1.9). 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 (13.3) functions. A union shall not have base classes. A union shall not be used as a base class. If a union contains a non-static data member of reference type the program is ill-formed. [Note: Absent default member initializers (12.2), if any non-static data member of a union has a non-trivial default constructor (15.1), copy constructor (15.8), move constructor (15.8), copy assignment operator (15.8), move assignment operator (15.8), or destructor (15.4), the corresponding member function of the union must be user-provided or it will be implicitly deleted (11.4.3) for the union. — end note]

[Example: Consider the following union:

```cpp
union U {
    int i;
    float f;
    std::string s;
};
```

Since `std::string` (24.3) 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 (8.5.1.5) that nominates a union member, it may begin the lifetime of that union member, as described below. For an expression `E`, define the set $S(E)$ of subexpressions of `E` as follows:

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 (8.5.18) or a trivial assignment operator (15.8), 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] [Example:

```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;
}
```
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 (12.2)  
    int n = y.x.a;  // OK: ends lifetime of y.x, y.k is active member of union  
    y.k = 4;  // undefined behavior: y.x.b modified outside its lifetime,  
    y.x.b = n;  // S(y.x.b) is empty because X's default constructor is deleted,  
    // so union member y.x's lifetime does not implicitly start  
}  

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]  

12.3.1 Anonymous unions  

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; };  
            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 (Clause 14). 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 11.6.1. — end note]
A union-like class is a union or a class that has an anonymous union as a direct member. A union-like class X has a set of variant members. If X is a union, a non-static data member of X that is not an anonymous union is a variant member of X. In addition, a non-static data member of an anonymous union that is a member of X is also a variant member of X. At most one variant member of a union may have a default member initializer. [Example:

```c
union U {
    int x = 0;
    union {
        int k;
    };
    union {
        int z;
        int y = 1;  // error: initialization for second variant member of U
    };
};
```
—end example]

### 12.4 Local class declarations [class.local]

A class can be declared within a function definition; such a class is called a local class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same access to names outside the function as does the enclosing function. [Note: A declaration in a local class cannot odr-use (6.2) a local entity from an enclosing scope. —end note] [Example:

```c
int x;
void f() {
    static int s;
    int x;
    const int N = 5;
    extern int q();
    int arr[2];
    auto [y, z] = arr;

    struct local {
        int g() { return x; }  // error: odr-use of non-odr-usable variable x
        int h() { return s; }  // OK
        int k() { return ::x; }  // OK
        int l() { return q(); }  // OK
        int m() { return N; }  // OK: not an odr-use
        int* n() { return &N; }  // error: odr-use of non-odr-usable variable N
        int p() { return y; }  // error: odr-use of non-odr-usable structured binding y
    };

    local* p = 0;  // error: local not in scope

    —end example]

An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 14). Member functions of a local class shall be defined within their class definition, if they are defined at all.

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.

A local class shall not have static data members.
13 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  ...
  ,  base-specifier  ...

base-specifier:
  attribute-specifier-seq  class-or-decltype
  attribute-specifier-seq  virtual  access-specifier  class-or-decltype
  attribute-specifier-seq  access-specifier  virtual  class-or-decltype

class-or-decltype:
  nested-name-specifier  class-name
  nested-name-specifier  template  simple-template-id
decltype-specifier

access-specifier:
  private
  protected
  public
```

The optional attribute-specifier-seq appertains to the base-specifier.

A class-or-decltype shall denote a class type that is not an incompletely defined class (Clause 12). 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). If the name found is not a class-name, the program is ill-formed. 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 Clause 14 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 10.3.3. 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 (13.2). [Note: The scope resolution operator :: (8.4) 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 14.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (7.11). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (11.6.3). — end note]

The base-specifier-list specifies the type of the base class subobjects contained in an object of the derived class type. [Example:

```
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 (17.6.3).
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".]

![ Directed acyclic graph ]

Figure 2 — Directed acyclic graph

Note: Initialization of objects representing base classes can be specified in constructors; see 15.6.2. — end note]

Note: A base class subobject might have a layout (6.6.4) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (15.7) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause 12); 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 (8.5.10). — end note

13.1 Multiple base classes [class.mi]

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:

class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
— end example]

Note: The order of derivation is not significant except as specified by the semantics of initialization by constructor (15.6.2), cleanup (15.4), and storage layout (12.2, 14.1). — end note

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:

class X { /* ... */ };
class Y : public X, public X { /* ... */ }; // ill-formed
class L { public: int next; /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ }; // well-formed
class D : public A, public L { void f(); /* ... */ }; // well-formed
— end example]

A base class specifier that does not contain the keyword virtual specifies a non-virtual base class. A base class specifier that contains the keyword virtual specifies a virtual base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (6.6.2) 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. [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.

§ 13.1
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
do C::f() { A::next = B::next; }  // well-formed
```

Without the A:: or B:: qualifiers, the definition of C::f above would be ill-formed because of ambiguity (13.2).

— end note

[Note: In contrast, consider the case with a virtual base class:

```cpp
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

For an object c of class type C, a single subobject of type V is shared by every base class subobject of c that has a virtual base class of type V. Given the class C defined above, an object of class C will have one subobject of class V, as shown in Figure 4.

— end note

[Note: A class can have both virtual and non-virtual base classes of a given type.

```cpp
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

For an object of class AA, all virtual occurrences of base class B in the class lattice of AA correspond to a single B subobject within the object of type AA, and every other occurrence of a (non-virtual) base class B in the class lattice of AA corresponds one-to-one with a distinct B subobject within the object of type AA. Given the class AA defined above, class AA has two subobjects of class B: Z’s B and the virtual B shared by X and Y, as shown in Figure 5.

— end note

13.2 Member name lookup

[class.member.lookup]

1 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 id-expression, 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, Clause 14).

2 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 (10.3.3), and type declarations (including injected-class-names) are replaced by the types they designate. \( S(f,C) \) is calculated as follows:

1. If \( C \) contains a declaration of the name \( f \), the declaration set contains every declaration of \( f \) declared in \( C \) that satisfies the requirements of the language construct in which the lookup occurs. [Note: Looking up a name in an *elaborated-type-specifier* (6.4.4) or *base-specifier* (Clause 13), 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* (10.3.3) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name in the same scope. — end note] If the resulting declaration set is not empty, the subobject set contains \( C \) itself, and calculation is complete.

2. Otherwise (i.e., \( C \) does not contain a declaration of \( f \) or the resulting declaration set is empty), \( S(f,C) \) is initially empty. If \( C \) has base classes, calculate the lookup set for \( f \) in each direct base class subobject \( B_i \), and merge each such lookup set \( S(f,B_i) \) in turn into \( S(f,C) \).

3. The following steps define the result of merging lookup set \( S(f,B_i) \) into the intermediate \( S(f,C) \):

   (6.1) If each of the subobject members of \( S(f,B_i) \) is a base class subobject of at least one of the subobject members of \( S(f,C) \), or if \( S(f,B_i) \) is empty, \( S(f,C) \) is unchanged and the merge is complete. Conversely, if each of the subobject members of \( S(f,C) \) is a base class subobject of at least one of the subobject members of \( S(f,B_i) \), or if \( S(f,C) \) is empty, the new \( S(f,C) \) is a copy of \( S(f,B_i) \).

   (6.2) Otherwise, if the declaration sets of \( S(f,B_i) \) and \( S(f,C) \) differ, the merge is ambiguous: the new \( S(f,C) \) is a lookup set with an invalid declaration set and the union of the subobject sets. In subsequent merges, an invalid declaration set is considered different from any other.

   (6.3) Otherwise, the new \( S(f,C) \) is a lookup set with the shared set of declarations and the union of the subobject sets.

4. The result of name lookup for \( f \) in \( C \) is the declaration set of \( S(f,C) \). If it is an invalid set, the program is ill-formed. [Example:

```
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
}
```

5. \( S(x,F) \) 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]

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:

```
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
}
```

6. If the name of an overloaded function is unambiguously found, overload resolution (16.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [Example:
struct A {
  int f();
};
struct B {
  int f();
};
struct C : A, B {
  int f() { return A::f() + B::f(); }
};

—end example

[Note: A static member, a nested type or an enumerator defined in a base class T can unambiguously be found even if an object has more than one base class subobject of type T. Two base class subobjects share the non-static member subobjects of their common virtual base classes. —end note] [Example:

struct V {
  int v;
};
struct A {
  int a;
  static int s;
  enum { e };
};
struct B : A, virtual V {
};
struct C : A, virtual V {
};
struct D : B, C {

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 as in D
}
—end example]

[Note: When virtual base classes are used, a hidden declaration can be reached along a path through the subobject lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. —end note] [Example:

struct V { int f(); int x; }; struct W { int g(); int y; }; struct B : virtual V, W {
  int f(); int x;
  int g(); int y;
}; struct C : virtual V, W {

struct D : B, C { void glorp(); };

![Figure 6 — Name lookup](image)

The names declared in V and the left-hand instance of W are hidden by those in B, but the names declared in the right-hand instance of W are not hidden at all.
void D::glorp() {
    x++; // OK: B::x hides V::x
    f(); // OK: B::f() hides V::f()
    y++; // error: B::y and C's W::y
    g(); // error: B::g() and C's W::g()
}

—end example—

An explicit or implicit conversion from a pointer to or an expression designating an object of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [ Example:

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 (7.12, 8.5.1.5, 14.2). —end note—

Example:

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—

13.3 Virtual functions [class.virtual]

[ 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.

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 (11.3.5), cv-qualification, and ref-qualifier (or absence of same) as Base::vf is declared, then Derived::vf is also virtual (whether or not it is so declared) and it overrides Base::vf. For convenience we say that any virtual function overrides itself.

A function with the same name but a different parameter list (Clause 16) as a virtual function is not necessarily virtual and does not override. The use of the virtual specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (Clause 14) is not considered in determining overriding.
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:

```c++
struct A {
    virtual void f();
};
struct B : virtual A {
    virtual void f();
};
struct C : B, virtual A {
    using A::f;
};

void foo() {
    C c;
    c.f(); // calls B::f, the final overrider
    c.C::f(); // calls A::f because of the using-declaration
}
@end example]

[Example:

```c++
struct A { virtual void f(); };
struct B : A { };
struct C : A { void f(); };
struct D : B, C { };// OK: A::f and C::f are the final overrides
    // 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,

```c++
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 \texttt{virt-specifier final} and in a class \( D \) derived from \( B \) a function \( D::f \) overrides \( B::f \), the program is ill-formed. [Example:

```c++
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 \texttt{virt-specifier override} and does not override a member function of a base class, the program is ill-formed. [Example:

```c++
struct B {
    virtual void f(int);
};
```
A virtual function shall not have a trailing requires-clause (Clause 11). [Example:

```c++
struct A {
  virtual void f() requires true; // error: virtual function cannot be constrained (17.4.2)
};
```

—end example]  

Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see 15.4 and 15.5.

The return type of an overriding function shall be either identical to the return type of the overridden function or covariant with the classes of the functions. If a function D::f overrides a function B::f, the return types of the functions are covariant if they satisfy the following criteria:

1. Both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes.
2. The class in the return type of B::f is the same class as the class in the return type of D::f, or is an unambiguous and accessible direct or indirect base class of the class in the return type of D::f.
3. Both pointers or references have the same cv-qualification and the class type in the return type of D::f has the same cv-qualification as or less cv-qualification than the class type in the return type of B::f.

If the class type in the covariant return type of D::f differs from that of B::f, the class type in the return type of D::f shall be complete at the point of declaration of D::f or shall be the class type D. When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (8.5.1.2). [Example:

```c++
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
  A* vf4(); // OK: returns pointer to derived class
  A* vf5(); // error: returns pointer to incomplete class
  void f();
};

void g() {
  Derived d;
  Base* bp = &d; // standard conversion:
  // Derived* to Base*
  bp->vf1(); // calls Derived::vf1()
  bp->vf2(); // calls Base::vf2()
}
```

113) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.
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

10 [ 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) (8.5.1.2). —end note]

11 [ Note: The virtual specifier implies membership, so a virtual function cannot be a non-member (10.1.2)
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 (14.3) in another class. —end note]

12 A virtual function declared in a class shall be defined, or declared pure (13.4) in that class, or both; no
diagnostic is required (6.2).

13 [ Example: Here are some uses of virtual functions with multiple base classes:

```c
struct A {
    virtual void f();
};

struct B1 : A {
    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]

14 [ Example: The following example shows a function that does not have a unique final overrider:

```c
struct A {
    virtual void f();
};

struct VB1 : virtual A {
    void f();
};

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

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

Explicit qualification with the scope operator (8.4) 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]

A function with a deleted definition (11.4) 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.

13.4 Abstract classes

1 [ 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]

2 An abstract class is a class that 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. A class is abstract if it has at least one pure virtual function. [Note: Such a function might be inherited: see below. —end note] A virtual function is specified pure by using a pure-specifier (12.2) in the function declaration in the class definition. A pure virtual function need be defined only if called with, or as if with (15.4), the qualified-id syntax (8.4).

[Example:

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:

struct C {
    virtual void f() = 0 { };     // ill-formed
};

—end example]
An abstract class shall not be used as a parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class can be declared. [Example:

```cpp
shape x; // error: object of abstract class
shape* p; // OK
shape f(); // error
void g(shape); // error
shape& h(shape&); // OK
```
—end example]

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
    };
```
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) {} // a definition is required somewhere
    void draw();
};
```
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 (13.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.
14 Member access control

1 A member of a class can be

(1.1) — private; that is, its name can be used only by members and friends of the class in which it is declared.

(1.2) — protected; that is, its name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends (see 14.4).

(1.3) — public; that is, its name can be used anywhere without access restriction.

2 A member of a class can also access all the names to which the class has access. A local class of a member function may access the same names that the member function itself may access.

3 Members of a class defined with the keyword `class` are private by default. Members of a class defined with the keywords `struct` or `union` are public by default. [Example:

```cpp
class X {
  int a;  // X::a is private by default
};

struct S {
  int a;  // S::a is public by default
};
```
—end example]

4 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 (14.3) and using-declarations (10.3.3). —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 by default
  public:
    typedef B BB;
};

void f() {
  A::BB x;  // OK, typedef name A::BB is public
  A::B y;  // access error, A::B is private
}
```
—end note]

5 It should be noted that it is access to members and base classes that is controlled, not their visibility. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

6 All access controls in Clause 14 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]

7 [Example:

```cpp
class A {
  typedef int I;  // private member
  I f();
```
—end example]

114) Access permissions are thus transitive and cumulative to nested and local classes.
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

8 The names in a default argument (11.3.6) 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 17.8.1.

9 The names in a default template-argument (17.1) have their access checked in the context in which they appear rather than at any points of use of the default template-argument. [Example:

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]

14.1 Access specifiers [class.access.spec]

1 Member declarations can be labeled by an access-specifier (Clause 13):

    access-specifier : member-specification_opt

An access-specifier specifies the access rules for members following it until the end of the class or until another access-specifier is encountered. [Example:

class X {
    int a; // X::a is private by default: class used
    public:
    int b; // X::b is public
    int c; // X::c is public
};
—end example]

2 Any number of access specifiers is allowed and no particular order is required. [Example:

struct S {
    int a; // S::a is public by default: struct used
    protected:
    int b; // S::b is protected
    private:
    int c; // S::c is private
}
14.2 Accessibility of base classes and base class members

If a class is declared to be a base class (Clause 13) 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.\(^{115}\)

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:

```c
class B { /* ... */ };  
class D1 : private B { /* ... */ };  
class D2 : public B { /* ... */ };  
class D3 : B { /* ... */ };  
struct D4 : public B { /* ... */ };  
struct D5 : private B { /* ... */ };  
struct D6 : B { /* ... */ };  
struct D7 : protected B { /* ... */ };  
struct D8 : protected B { /* ... */ };  
```

Here B is a public base of D2, D4, and D6, a private base of D1, D3, and D5, and a protected base of D7 and D8. — end example]\(^{115}\)

[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 (7.11) and explicit casts (8.5.3), 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,

---

\(^{115}\) As specified previously in Clause 14, 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.
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; // OK with cast
    bp2->mi = 3;   // OK: access through a pointer to B.
}

—end note

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 (7.11, 7.12). [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 (8.5.1.5) is used (including cases where an implicit “this->” is added). If both a class member access operator and a qualified-id are used to name the member (as in p->T::m), the class naming the member is the class denoted by the nested-name-specifier of the qualified-id (that is, \(T\)). — end note] A member \(m\) is accessible at the point \(R\) when named in class \(N\) if
— m as a member of N is public, or
— m as a member of N is private, and R occurs in a member or friend of class N, or
— 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
— there exists a base class B of N that is accessible at R, and m is accessible at R when named in class B.

\[\text{Example:}\]

\begin{verbatim}
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{verbatim}

— end example \]  

6 If a class member access operator, including an implicit “\texttt{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
“\texttt{.}” operator case) cannot be implicitly converted to a pointer to the naming class of the right operand. \[\text{Note:}\] This requirement is in addition to the requirement that the member be accessible as named. — end note\]

\section*{14.3 Friends}

1 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. \[\text{Example:}\] The following example illustrates the differences between members and friends:

\begin{verbatim}
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 \]

2 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 \textit{base-specifiers} and member declarations of the befriended class.
\[\text{Example:}\]

\begin{verbatim}
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{verbatim}
3 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 17, 17.6.4). — 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:

   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 (10.1.1).

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:

   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 (12.4), 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]
```

Such a function is implicitly an inline function (10.1.6). 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).

No `storage-class-specifier` shall appear in the `decl-specifier-seq` of a friend declaration.

A name nominated by a friend declaration shall be accessible in the scope of the class containing the friend declaration. The meaning of the friend declaration is the same whether the friend declaration appears in the private, protected, or public (12.2) portion of the class `member-specification`.

Friendship is neither inherited nor transitive. [Example:

```cpp
class A {
  friend class B;
  int a;
};
class B {
  friend class C;
};
class C {
  void f(A* p) {
    p->a++;
    // error: C is not a friend of A despite being a friend of a friend
  }
};
class D : public B {
  void f(A* p) {
    p->a++;
    // error: D is not a friend of A despite being derived from a friend
  }
};
—end example]
```

If a friend declaration appears in a local class (12.4) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope. For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing non-class scope. [Example:

```cpp
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
    friend class X;  // OK, but X is a local class, not ::X
    friend class Y;  // OK
    friend class Z;  // OK, introduces local class Z
    friend void a(); // error, ::a is not considered
    friend void b(); // OK
    friend void c(); // error
  };
  X* px;  // OK, but ::X is found
  Z* pz;  // error, no ::Z is found
}
—end example]
```
14.4 Protected member access

An additional access check beyond those described earlier in Clause 14 is applied when a non-static data member or non-static member function is a protected member of its naming class (14.2). As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class C. If the access is to form a pointer to member (8.5.2.1), the nested-name-specifier shall denote C or a class derived from C. All other accesses involve a (possibly implicit) object expression (8.5.1.5). In this case, the class of the object expression shall be C or a class derived from C. [Example:

```cpp
class B {
  protected:
    int i;
    static int j;
};
class D1 : public B {
};
class D2 : public B {
  friend void fr(B*, D1*, D2*);
  void mem(B*, D1*);
};
void fr(B* pb, D1* p1, D2* p2) {
  pb->i = 1; // ill-formed
  p1->i = 2; // ill-formed
  p2->i = 3; // OK (access through a D2)
  p2->B::i = 4; // OK (access through a D2, even though naming class is B)
  int B::* pmi_B = &B::i; // ill-formed
  int B::* pm_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::* pm_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]

14.5 Access to virtual functions

The access rules (Clause 14) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. [Example:

```cpp
class B {
  public:
    virtual int f();
};
```

116) This additional check does not apply to other members, e.g., static data members or enumerator member constants.
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]

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.

14.6 Multiple access
[14.7 246]

1 If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [Example:

```cpp
class W { public: void f(); }
class A : private virtual W { }
class B : public virtual W { }
class C : public A, public B {
  void f() { W::f(); }    // OK
}
```

Since W::f() is available to C::f() along the public path through B, access is allowed. —end example]  

14.7 Nested classes
[14.7 246]

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 (Clause 14) shall be obeyed. [Example:

```cpp
class E {
  int x;
  class B { }

  class I {
    B b;    // OK: E::I can access E::B
    int y;
    void f(E* p, int i) {
      p->x = i;    // OK: E::I can access E::x
    }
  }

  int g(I* p) {
    return p->y;    // error: I::y is private
  }
};

—end example]

§ 14.7 246
15 Special member functions

The default constructor (15.1), copy constructor and copy assignment operator (15.8), move constructor and move assignment operator (15.8), and destructor (15.4) are special member functions. [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 (8.6). See 15.1, 15.4 and 15.8. — 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.

Programs may explicitly refer to implicitly-declared special member functions. [Example: A program may explicitly call, take the address of, or form a pointer to member to an implicitly-declared special member function.]

[Note: The special member functions affect the way objects of class type are created, copied, moved, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. — end note]

Special member functions obey the usual access rules (Clause 14). [Example: Declaring a constructor protected ensures that only derived classes and friends can create objects using it. — end example]

For a class, its non-static data members, its non-virtual direct base classes, and, if the class is not abstract (13.4), its virtual base classes are called its potentially constructed subobjects.

15.1 Constructors

Constructors do not have names. In a declaration of a constructor, the declarator is a function declarator (11.3.5) 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. in a member-declaration that belongs to the member-specification of a class but is not a friend declaration (14.3), the id-expression is the injected-class-name (Clause 12) of the immediately-enclosing class;
2. in a member-declaration that belongs to the member-specification of a class template but is not a friend declaration, the id-expression is a class-name that names the current instantiation (17.7.2.1) of the immediately-enclosing class template; or
3. 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).

The class-name shall not be a typedef-name. In a constructor declaration, each declspecifier in the optional declspecifier-seq shall be friend, inline, explicit, or constexpr. [Example:

\[
\text{struct S { }
\]

\[
S(); \quad \text{// declares the constructor}
\]

\[
\text{S::S() { }} \quad \text{// defines the constructor}
\]

— end example]
A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (8.5.1.3) will cause a constructor to be called to initialize an object. [Note: For initialization of objects of class type see 15.6. — end note]

A constructor can be invoked for a \texttt{const}, \texttt{volatile} or \texttt{const volatile} object. \texttt{const} and \texttt{volatile} semantics (10.1.7.1) 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 \textit{default} constructor for a class \texttt{X} is a constructor of class \texttt{X} for which each parameter that is not a function parameter pack has a default argument (including the case of a constructor with no parameters). If there is no user-declared constructor for class \texttt{X}, a non-explicit constructor having no parameters is implicitly declared as defaulted (11.4). An implicitly-declared default constructor is an inline public member of its class.

A defaulted default constructor for class \texttt{X} is defined as deleted if:

\begin{itemize}
  \item \texttt{X} is a union that has a variant member with a non-trivial default constructor and no variant member of \texttt{X} has a default member initializer,
  \item \texttt{X} is a non-union class that has a variant member \texttt{M} with a non-trivial default constructor and no variant member of the anonymous union containing \texttt{M} has a default member initializer,
  \item any non-static data member with no default member initializer (12.2) is of reference type,
  \item any non-variant non-static data member of const-qualified type (or array thereof) with no \textit{brace-or-equal-initializer} does not have a user-provided default constructor,
  \item \texttt{X} is a union and all of its variant members are of const-qualified type (or array thereof),
  \item \texttt{X} is a non-union class and all members of any anonymous union member are of const-qualified type (or array thereof),
  \item any potentially constructed subobject, except for a non-static data member with a \textit{brace-or-equal-initializer}, has class type \texttt{M} (or array thereof) and either \texttt{M} has no default constructor or overload resolution (16.3) as applied to find \texttt{M}'s corresponding constructor results in an ambiguity or in a function that is deleted or inaccessible from the defaulted default constructor, or
  \item any potentially constructed subobject has a type with a destructor that is deleted or inaccessible from the defaulted default constructor.
\end{itemize}

A default constructor is \textit{trivial} if it is not user-provided and if:

\begin{itemize}
  \item its class has no virtual functions (13.3) and no virtual base classes (13.1), and
  \item no non-static data member of its class has a default member initializer (12.2), and
  \item all the direct base classes of its class have trivial default constructors, and
  \item 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.
\end{itemize}

Otherwise, the default constructor is \textit{non-trivial}.

A default constructor that is defaulted and not defined as deleted is \textit{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 (8.6), 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 \textit{ctor-initializer} (15.6.2) and an empty \textit{compound-statement}. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a \texttt{constexpr} constructor (10.1.5), the implicitly-defined default constructor is \texttt{constexpr}. Before the defaulted default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [Note: An implicitly-declared default constructor has an exception specification (18.4). An explicitly-defaulted definition might have an implicit exception specification, see 11.4. — end note]

Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (6.6.4.1, 6.6.4.2, 6.6.4.3) defined without an initializer (11.6), are called to create class objects of dynamic storage duration (6.6.4.4) created by a \texttt{new-expression} in which the \texttt{new-initializer} is omitted (8.5.2.4), or are called when the explicit type conversion syntax (8.5.1.3) is used. A program is ill-formed if the default constructor for an object is implicitly used and the constructor is not accessible (Clause 14).
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 functional notation type conversion (8.5.1.3) can be used to create new objects of its type. [Note: The syntax looks like an explicit call of the constructor. — end note]  

Example:
```cpp
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

—end example—

An object created in this way is unnamed. [Note: 15.2 describes the lifetime of temporary objects. — end note] [Note: Explicit constructor calls do not yield lvalues, see 8.2.1. — end note]

[Note: Some language constructs have special semantics when used during construction; see 15.6.2 and 15.7. — end note]

During the construction of an object, if the value of the object or any of its subobjects is accessed through a glvalue that is not obtained, directly or indirectly, from the constructor’s this pointer, the value of the object or subobject thus obtained is unspecified. [Example:
```cpp
struct C;
void no_opt(C*);

struct C {
  int c;
  C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cptr) {
  int i = cobj.c * 100;
  cptr->c = 1;
  cout << cobj.c * 100
       << '\n';
}

extern struct D d;
struct D {
  D(int a) : a(a), b(d.a) {}
  int a, b;
};
D d = D(1);
```

—end example—]

15.2 Temporary objects

Temporary objects are created

(1.1) when a prvalue is materialized so that it can be used as a glvalue (7.4),

(1.2) when needed by the implementation to pass or return an object of trivially-copyable type (see below), and

(1.3) when throwing an exception (18.1). [Note: The lifetime of exception objects is described in 18.1. — end note]

Even when the creation of the temporary object is unevaluated (8.2), all the semantic restrictions shall be respected as if the temporary object had been created and later destroyed. [Note: This includes accessibility (Clause 14) 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 (8.5.1.2), no temporary is introduced, so the foregoing does not apply to such a prvalue. — end note]

The materialization of a temporary object is generally delayed as long as possible in order to avoid creating unnecessary temporary objects. [Note: Temporary objects are materialized:
— when binding a reference to a prvalue (11.6.3, 8.5.1.3, 8.5.1.7, 8.5.1.9, 8.5.1.11, 8.5.3),

— when performing member access on a class prvalue (8.5.1.5, 8.5.4),

— when performing an array-to-pointer conversion or subscripting on an array prvalue (7.2, 8.5.1.1),

— when initializing an object of type `std::initializer_list<T>` from a braced-init-list (11.6.4),

— for certain unevaluated operands (8.5.1.8, 8.5.2.3), and

— when a prvalue appears as a discarded-value expression (8.2).

— end note

Example: Consider the following code:

```cpp
class X {
public:
  X(int);
  X(const X&);
  X& operator=(const X&);
  ~X();
};
class Y {
public:
  Y(int);
  Y(Y&&);
  ~Y();
};
X f(X);
Y g(Y);

void h() {
  X a(1);
  X b = f(X(2));
  Y c = g(Y(3));
  a = f(a);
}
```

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

When an object of class type X is passed to or returned from a function, if each copy constructor, move constructor, and destructor of X is either trivial or deleted, and 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). [Note: This latitude is granted to allow objects of class type to be passed to or returned from functions in registers. — end note]

When an implementation introduces a temporary object of a class that has a non-trivial constructor (15.1, 15.8), 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 (15.4). 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.

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 (11.6). The second context is when a copy constructor is called to copy an element of an array while the entire array is copied (8.4.5.2, 15.8). 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:

1. A temporary materialization conversion (7.4),
2. (expression), where expression is one of these expressions,
3. subscripting (8.5.1.1) of an array operand, where that operand is one of these expressions,
4. A class member access (8.5.1.5) using the . operator where the left operand is one of these expressions and the right operand designates a non-static data member of non-reference type,
5. A pointer-to-member operation (8.5.4) 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,
6. A
   1. const_cast (8.5.1.11),
   2. static_cast (8.5.1.9),
   3. dynamic_cast (8.5.1.7), or
   4. reinterpret_cast (8.5.1.10) converting, without a user-defined conversion, a glvalue operand that is one of these expressions to a glvalue that refers to the object designated by the operand, or to its complete object or a subobject thereof,
7. A conditional expression (8.5.16) that is a glvalue where the second or third operand is one of these expressions, or
8. A comma expression (8.5.19) that is a glvalue where the right operand is one of these expressions.

Example:

```cpp
template< typename T> using id = T;

int&& a = id<int[3]>{1, 2, 3}[i]; // temporary array has same lifetime as a
const int& b = static_cast<const int&>(0); // temporary int has same lifetime as b
int&& c = cond ? id<int[3]>{1, 2, 3}[i] : static_cast<int&&>(0);
// exactly one of the two temporaries is lifetime-extended
```

Note: An explicit type conversion (8.5.1.3, 8.5.3) 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
```

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{1}; // both S and int temporaries have lifetime of s
```

The exceptions to this lifetime rule are:

9. A temporary object bound to a reference parameter in a function call (8.5.1.2) persists until the completion of the full-expression containing the call.
10. The lifetime of a temporary bound to the returned value in a function return statement (9.6.3) is not extended; the temporary is destroyed at the end of the full-expression in the return statement.
11. A temporary bound to a reference in a new-initializer (8.5.2.4) persists until the completion of the full-expression containing the new-initializer. Note: This may introduce a dangling reference.

Note: The same rules apply to initialization of an initializer_list object (11.6.4) with its underlying temporary array.

117) The same rules apply to initialization of an initializer_list object (11.6.4) with its underlying temporary array.
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 (6.6.4.1, 6.6.4.2, 6.6.4.3); 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.

Example:

class 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]

15.3 Conversions [class.conv]

1 Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called user-defined conversions and are used for implicit type conversions (Clause 7), for initialization (11.6), and for explicit type conversions (8.5.3, 8.5.1.9).

2 User-defined conversions are applied only where they are unambiguous (13.2, 15.3.2). Conversions obey the access control rules (Clause 14). Access control is applied after ambiguity resolution (6.4).

3 [Note: See 16.3 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...
Function overload resolution (16.3.3) selects the best conversion function to perform the conversion.

*Example:*

```cpp
struct X {
    operator int();
};

struct Y : X {
    operator char();
};

void f(Y& a) {
    if (a) {
        // ill-formed: X::operator int() or Y::operator char()
    }
}
```

15.3.1 Conversion by constructor

A constructor declared without the `function-specifier explicit` specifies a conversion from the types of its parameters (if any) to the type of its class. Such a constructor is called a **converting constructor**.  

*Example:*

```cpp
struct X {
    X(int);
    X(const char*, int = 0);
    X(int, int);
};

void f(X arg) {
    X a = 1;  // a = X(1)
    X b = "Jessie";  // b = X("Jessie", 0)
    a = 2;  // a = X(2)
    f(3);  // f(X(3))
    f({1, 2});  // f(X(1,2))
}
```

1

Note: An explicit constructor constructs objects just like non-explicit constructors, but does so only where the direct-initialization syntax (11.6) or where casts (8.5.1.9, 8.5.3) are explicitly used; see also 16.3.1.4. A default constructor may be an explicit constructor; such a constructor will be used to perform default-initialization or value-initialization (11.6).  

*Example:*

```cpp
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
```

2

Note: An implicitly-declared copy/move constructor is not an explicit constructor; it may be called for implicit type conversions. — end note

A non-explicit copy/move constructor (15.8) 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

§ 15.3.1
15.3.2 Conversion functions

A member function of a class \( X \) having no parameters with a name of the form

\[
\text{conversion-function-id:}
\]

\[
\text{operator conversion-type-id}
\]

\[
\text{conversion-type-id:}
\]

\[
type-specifier-seq\ conversion-declarator_{opt}
\]

\[
\text{conversion-declarator:}
\]

\[
\text{ptr-operator conversion-declarator}_{opt}
\]

specifies a conversion from \( X \) to the type specified by the \( \text{conversion-type-id} \). Such functions are called conversion functions. A \( \text{decl-specifier} \) in the \( \text{decl-specifier-seq} \) of a conversion function (if any) shall be neither a \( \text{defining-type-specifier} \) nor \( \text{static} \). The type of the conversion function (11.3.5) is “function taking no parameter returning \( \text{conversion-type-id} \)”. 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.\(^\text{118}\) [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 \( X::\text{operator int()} \). —end example]

A conversion function may be explicit (10.1.2), in which case it is only considered as a user-defined conversion for direct-initialization (11.6). 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]

The \( \text{conversion-type-id} \) shall not represent a function type nor an array type. The \( \text{conversion-type-id} \) in a \( \text{conversion-function-id} \) is the longest sequence of tokens that could possibly form a \( \text{conversion-type-id} \). [Note: This prevents ambiguities between the declarator operator * and its expression counterparts. [Example:

```c
&ac.operator int*i; // syntax error;
// parsed as: &\( \text{ac.operator int \ast} \)i
// not as: &\( \text{ac.operator int} \ast \)i
```

118] These conversions are considered as standard conversions for the purposes of overload resolution (16.3.3.1, 16.3.3.1.4) and therefore initialization (11.6) and explicit casts (8.5.1.9). A conversion to \( \text{void} \) does not invoke any conversion function (8.5.1.9). Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class.
The * is the pointer declarator and not the multiplication operator. — end example] This rule also prevents ambiguities for attributes. [Example:

```c
operator int [(noreturn)] ();  // error: noreturn attribute applied to a type
```
— end example] — end note]

Conversion functions are inherited.

Conversion functions can be virtual.

A conversion function template shall not have a deduced return type (10.1.7.4). [Example:

```c
struct S {
    operator auto() const { return 10; }  // OK
    template<class T>
    operator auto() const { return 1.2; }  // error: conversion function template
};
```
— end example]

### 15.4 Destructors

1 In a declaration of a destructor, the declarator is a function declarator (11.3.5) of the form

```
ptr-declarator ( parameter-declaration-clause ) noexcept-specifier_opt attribute-specifier-seq_opt
```

where the ptr-declarator consists solely of an id-expression, an optional attribute-specifier-seq, and optional surrounding parentheses, and the id-expression has one of the following forms:

1.1 — in a member-declaration that belongs to the member-specification of a class but is not a friend declaration (14.3), the id-expression is ~class-name and the class-name is the injected-class-name (Clause 12) of the immediately-enclosing class;

1.2 — in a member-declaration that belongs to the member-specification of a class template but is not a friend declaration, the id-expression is ~class-name and the class-name names the current instantiation (17.7.2.1) of the immediately-enclosing class template; or

1.3 — in a declaration at namespace scope or in a friend declaration, the id-expression is nested-name-specifier ~class-name and the class-name names the same class as the nested-name-specifier.

The class-name shall not be a typedef-name. A destructor shall take no arguments (11.3.5). Each decl-specifier of the decl-specifier-seq of a destructor declaration (if any) shall be friend, inline, or virtual.

2 A destructor is used to destroy objects of its class type. The address of a destructor shall not be taken. A destructor can be invoked for a const, volatile or const volatile object. const and volatile semantics (10.1.7.1) 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.

3 [Note: A declaration of a destructor that does not have a noexcept-specifier has the same exception specification as if had been implicitly declared (18.4). — end note]

4 If a class has no user-declared destructor, a destructor is implicitly declared as defaulted (11.4). An implicitly-declared destructor is an inline public member of its class.

5 A defaulted destructor for a class X is defined as deleted if:

5.1 — X is a union-like class that has a variant member with a non-trivial destructor,

5.2 — any potentially constructed subobject has class type M (or array thereof) and M has a deleted destructor or a destructor that is inaccessible from the defaulted destructor,

5.3 — or, for a virtual destructor, lookup of the non-array deallocation function results in an ambiguity or in a function that is deleted or inaccessible from the defaulted destructor.

6 A destructor is trivial if it is not user-provided and if:

6.1 — the destructor is not virtual,

6.2 — all of the direct base classes of its class have trivial destructors, and

6.3 — for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

Otherwise, the destructor is non-trivial.
A destructor that is defaulted and not defined as deleted is 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 type of the most derived class (15.6.2), 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 15.6). A return statement (9.6.3) 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 15.6).

A destructor can be declared virtual (13.3) or pure virtual (13.4); 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.

A destructor is invoked implicitly

(12.1) — for a constructed object with static storage duration (6.6.4.1) at program termination (6.8.3.4),

(12.2) — for a constructed object with thread storage duration (6.6.4.2) at thread exit,

(12.3) — for a constructed object with automatic storage duration (6.6.4.3) when the block in which an object is created exits (9.7),

(12.4) — for a constructed temporary object when its lifetime ends (7.4, 15.2).

In each case, the context of the invocation is the context of the construction of the object. A destructor is also invoked implicitly through use of a delete-expression (8.5.2.5) for a constructed object allocated by a new-expression (8.5.2.4); the context of the invocation is the delete-expression. [Note: An array of class type contains several subobjects for each of which the destructor is invoked. — end note] A destructor can also be invoked explicitly. A destructor is potentially invoked if it is invoked or as specified in 8.5.2.4, 15.6.2, and 18.1. 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 (15.8)), the non-array deallocation function is determined as if for the expression delete this appearing in a non-virtual destructor of the destructor’s class (see 8.5.2.5). If the lookup fails or if the deallocation function has a deleted definition (11.4), the program is ill-formed. [Note: This assures that a deallocation function corresponding to the dynamic type of an object is available for the delete-expression (15.5). — end note]

In an explicit destructor call, the destructor is specified by a ~ followed by a type-name or decltype-specifier that denotes the destructor’s class type. The invocation of a destructor is subject to the usual rules for member functions (12.2.1); that is, if the object is not of the destructor’s class type and not of a class derived from the destructor’s class type (including when the destructor is invoked via a null pointer value), the program has undefined behavior. [Note: Invoking delete on a null pointer does not call the destructor; see 8.5.2.5. — end note] [Example:

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

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

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
    D_object.B::~B(); // calls B’s destructor
}
```
B_ptr->~B();  // calls B’s destructor
B_ptr->~B_alias(); // 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 (8.5.1.5) or a qualified-id (8.4); in particular, the unary-expression ~X() in a member function is not an explicit destructor call (8.5.2.1). —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,

```c
void* operator new(std::size_t, void* p) { return p; }
struct X {
  X(int);
  ~X();
};
void f(X* p);

void g() {  // rare, specialized use:
  char* buf = new char[sizeof(X)];
  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 (8.5.1.4). Allowing this makes it possible to write code without having to know if a destructor exists for a given type. For example:

typedef int I;
I* p;
p->I::~I();  
—end note]  

15.5  Free store [class.free]

Any allocation function for a class T is a static member (even if not explicitly declared static).

1  

[Example:

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]  

§ 15.5  257
When an object is deleted with a `delete-expression (8.5.2.5)` 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 (6.6.4.4.2).

Class-specific deallocation function lookup is a part of general deallocation function lookup (8.5.2.5) 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 (15.4). Otherwise, if the `delete-expression` is used to deallocate an object of class T or array thereof, the static and dynamic types of the object shall be identical and 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 (8.5.2.5) continues. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

Any deallocation function for a class X is a static member (even if not explicitly declared `static`). [Example:

```cpp
class X {
    void operator delete(void*);
    void operator delete[](void*, std::size_t);
};

class Y {
    void operator delete(void*, std::size_t);
    void operator delete[](void*);
};

-- end example]
```

Since member allocation and deallocation functions are static they cannot be virtual. [Note: However, when the `cast-expression` of a `delete-expression` refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same. For example,

```cpp
struct B {
    virtual ~B();
    void operator delete(void*, std::size_t);
};

struct D : B {
    void operator delete(void*);
};

void f() {
    B* bp = new D;
    delete bp; // 1: uses D::operator delete(void*)
}
```

Here, storage for the non-array object of class D is deallocated by `D::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];
```

119] A similar provision is not needed for the array version of `operator delete` because 8.5.2.5 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.
delete[] bp; // undefined behavior

--- end note

7 Access to the deallocation function is checked statically. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. [Example: For the call on line “// 1” above, if B::operator delete() had been private, the delete expression would have been ill-formed. — end example]

8 [Note: If a deallocation function has no explicit noexcept-specifier, it has a non-throwing exception specification (18.4). — end note]

15.6 Initialization

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 11.6.

2 An object of class type (or array thereof) can be explicitly initialized; see 15.6.1 and 15.6.2.

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 11.3.4. [Note: Destructors for the array elements are called in reverse order of their construction. — end note]

15.6.1 Explicit initialization

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 11.6. [Example:

```cpp
struct complex {
    complex();
    complex(double);
    complex(double,double);
};

complex sqrt(complex,complex);

complex a(1); // initialize by a call of complex(double)
complex b = a; // initialize by a copy of a
complex c = complex(1,2); // construct complex(1,2) using complex(double,double),
// copy/move it into c
complex d = sqrt(b,c); // call sqrt(complex,complex) and copy/move the result into d
complex e; // initialize by a call of complex()
complex f = 3; // construct complex(3) using complex(double), copy/move it into f
complex g = { 1, 2 }; // initialize by a call of complex(double, double)
```

—end example] [Note: Overloading of the assignment operator (16.5.3) has no effect on initialization. — end note]

2 An object of class type can also be initialized by a braced-init-list. List-initialization semantics apply; see 11.6 and 11.6.4. [Example:

```cpp
complex v[6] = { 1, complex(1,2), complex(), 2 };
```

Here, complex::complex(double) is called for the initialization of v[0] and v[3], complex::complex(double, double) is called for the initialization of v[1], complex::complex() is called for the initialization v[2], v[4], and v[5]. For another example,

```cpp
struct X {
    int i;
    float f;
    complex c;
} x = { 99, 88.8, 77.7 }; 
```

Here, x.i is initialized with 99, x.f is initialized with 88.8, and complex::complex(double) is called for the initialization of x.c. — end example] [Note: Braces can be elided in the initializer-list for any aggregate,
even if the aggregate has members of a class type with user-defined type conversions; see 11.6.1. — end note]

3 [Note: If T is a class type with no default constructor, any declaration of an object of type T (or array thereof) is ill-formed if no initializer is explicitly specified (see 15.6 and 11.6). — 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 9.7. — end note]

15.6.2 Initializing bases and members

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 ctor-initializer, which has the form

\[
\text{ctor-initializer:} \\
: \text{mem-initializer-list}
\]

\[
\text{mem-initializer-list:} \\
\text{mem-initializer-list} \ldots \text{opt} \\
\text{mem-initializer} \ldots \text{opt}
\]

\[
\text{mem-initializer:} \\
\text{mem-initializer-id} (\text{expression-list}\text{opt}) \\
\text{mem-initializer-id} \text{braced-init-list}
\]

\[
\text{mem-initializer-id:} \\
\text{class-or-decltype} \\
\text{identifier}
\]

2 In a \text{mem-initializer-id} an initial unqualified \text{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 \text{mem-initializer-id} naming the member or base class and composed of a single identifier refers to the class member. A \text{mem-initializer-id} for the hidden base class may be specified using a qualified name. — end note] Unless the \text{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 \text{mem-initializer} is ill-formed.

3 A \text{mem-initializer-list} can initialize a base class using any \text{class-or-decltype} that denotes that base class type. [Example:

\[
\begin{align*}
\text{struct A} & \{ \text{A}(); \}; \\
\text{typedef} & \text{A global A;} \\
\text{struct B} & \{ \}; \\
\text{struct C: public A, public B} & \{ \text{C}(); \}; \\
\text{C: :C() : global A()} & \{ \} & // \text{mem-initializer for base A}
\end{align*}
\]

— end example]

4 If a \text{mem-initializer-id} is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the \text{mem-initializer} is ill-formed. [Example:

\[
\begin{align*}
\text{struct A} & \{ \text{A}(); \}; \\
\text{struct B: public virtual A} & \{ \}; \\
\text{struct C: public A, public B} & \{ \text{C}(); \}; \\
\text{C: :C() : A()} & \{ \} & // \text{ill-formed: which A?}
\end{align*}
\]

— end example]

5 A \text{ctor-initializer} may initialize a variant member of the constructor’s class. If a \text{ctor-initializer} specifies more than one \text{mem-initializer} for the same member or for the same base class, the \text{ctor-initializer} is ill-formed.

6 A \text{mem-initializer-list} can delegate to another constructor of the constructor’s class using any \text{class-or-decltype} that denotes the constructor’s class itself. If a \text{mem-initializer-id} designates the constructor’s class, it shall be the only \text{mem-initializer}; the constructor is a delegating constructor, and the constructor selected by the \text{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:

\[
\begin{align*}
\text{struct C} & \{ \\
\text{C (int )} & \{ \} & // \#1: \text{non-delegating constructor} \\
\text{C (42)} & \{ \} & // \#2: \text{delegates to \#1} \\
\text{C (char c) : C(42.0)} & \{ \} & // \#3: \text{ill-formed due to recursion with \#4}
\end{align*}
\]

§ 15.6.2 260
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 11.6 for direct-initialization. [Example:

```cpp
struct B1 { B1(int); /* ... */ }
struct B2 { B2(int); /* ... */ }
struct D : B1, B2 {
    int B1 b;
    const int c;
};
D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4) { /* ... */ }
D d(10);
```
—end example] [Note: The initialization performed by each mem-initializer constitutes a full-expression (6.8.1). Any expression in a mem-initializer is evaluated as part of the full-expression that performs the initialization. —end note] A mem-initializer where the mem-initializer-id denotes a virtual base class is ignored during execution of a constructor of any class that is not the most derived class.

A temporary expression bound to a reference member in a mem-initializer is ill-formed. [Example:

```cpp
struct A {
    A() : v(42) { } // error
    const int& v;
};
```
—end example]

In a non-delegating constructor, if a given potentially constructed subobject is not designated by a mem-initializer-id (including the case where there is no mem-initializer-list because the constructor has no ctor-initializer), then

1. if the entity is a non-static data member that has a default member initializer (12.2) and either
2. the constructor’s class is a union (12.3), and no other variant member of that union is designated by a mem-initializer-id or
3. 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,
   then
   1. the entity is initialized from its default member initializer as specified in 11.6;
   2. otherwise, if the entity is an anonymous union or a variant member (12.3.1), no initialization is performed;
   3. otherwise, the entity is default-initialized (11.6).

[Note: An abstract class (13.4) 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:
};
```
If a given non-static data member has both a default member initializer and a member-initializer, the initialization specified by the member-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
```

In a non-delegating constructor, the destructor for each potentially constructed subobject of class type is potentially invoked (15.4). [Note: This provision ensures that destructors can be called for fully-constructed subobjects in case an exception is thrown (18.2). — 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`.

2. Then, direct base classes are initialized in declaration order as they appear in the `base-specifier-list` (regardless of the order of the member-initializers).

3. Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the member-initializers).

4. Finally, the compound-statement of the constructor body is executed. [Note: The declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. — end note]

[Example:

```cpp
struct V {
    V();
    V(int);
};

struct A : virtual V {
    A();
    A(int);
};

struct B : virtual V {
    B();
    B(int);
};
```
struct C : A, B, virtual V {
  C();
  C(int);
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

V v(1);  // use V(int)
A a(2);  // use V(int)
B b(3);  // use V()
C c(4);  // use V()

—end example

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, 13.3) can be called for an object under construction. Similarly, an object under construction can be the operand of the typeid operator (8.5.1.8) or of a dynamic_cast (8.5.1.7). However, if these operations are performed in a ctor-initializer (or in a function called directly or indirectly from a ctor-initializer) before all the mem-initializers for base classes have completed, the program has undefined behavior. [Example:

```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]
17 [Note: 15.7 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]

18 A mem-initializer followed by an ellipsis is a pack expansion (17.6.3) 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]

15.6.3 Initialization by inherited constructor [class.inhctor.init]

1 When a constructor for type B is invoked to initialize an object of a different type D (that is, when the constructor was inherited (10.3.3)), initialization proceeds as if a defaulted default constructor were used to initialize the D object and each base class subobject from which the constructor was inherited, except that the B subobject is initialized by the invocation of the inherited constructor. The complete initialization is considered to be a single function call; in particular, the initialization of the inherited constructor’s parameters is sequenced before the initialization of any part of the D object. [Example:

```cpp
struct B1 {
    B1(int, ...) { }
};

struct B2 {
    B2(double) { }
};

int get();

struct D1 : B1 {
    using B1::B1;  // inherits B1(int, ...)
    int x;
    int y = get();
};

void test() {
    D1 d(2, 3, 4);  // OK: B1 is initialized by calling B1(2, 3, 4),
    // then d.x is default-initialized (no initialization is performed),
    // then d.y is initialized by calling get()

    D1 e;  // error: D1 has a deleted default constructor
}

struct D2 : B2 {
    using B2::B2;
    B1 b;
};

D2 f(1.0);  // error: B1 has a deleted default constructor
```

```cpp
struct W { W(int); };
struct X : virtual W { using W::W; X() = delete; };
struct Y : X { using X::X; };
struct Z : Y, virtual W { using Y::Y; }
Z z(0);  // OK: initialization of Y does not invoke default constructor of X
```

```cpp
template<class T> struct Log : T {
    using T::T;  // inherits all constructors from class T

    ~Log() { std::clog << "Destroying wrapper" << std::endl; }
};
```
Class template `Log` wraps any class and forwards all of its constructors, while writing a message to the standard log whenever an object of class `Log` is destroyed. — end example]

If the constructor was inherited from multiple base class subobjects of type `B`, the program is ill-formed. [Example:

```cpp
struct A { A(int); };  
struct B : A { using A::A; };  
struct C1 : B { using B::B; };  
struct C2 : B { using B::B; };  
struct D1 : C1, C2 {  
    using C1::C1;  
    using C2::C2;  
};  
struct V1 : virtual B { using B::B; };  
struct V2 : virtual B { using B::B; };  
struct D2 : V1, V2 {  
    using V1::V1;  
    using V2::V2;  
};  
D1 d1(0);  // 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(0) to initialize N’s base class,  
    // use M() to initialize O’s base class

— end example]

When an object is initialized by an inherited constructor, initialization of the object is complete when the initialization of all subobjects is complete.

### 15.7 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:

```cpp
struct X { int i; };  
struct Y : X { Y(); };  // non-trivial  
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;  // undefined, upcast to a base class type  
X xobj;  // definition of xobj

OK, X is a trivial class

For another example,

```cpp
struct W { int j; };  
```
struct X : public virtual W { }
struct Y {
    int* p;
    X x;
    Y() : p(&x.j) { // undefined, x is not yet constructed
    }
};
—end example

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:

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
    X(this) {} // defined: upon construction of X, C/B/D/A sublattice is fully constructed
};
—end example

Member functions, including virtual functions (13.3), can be called during construction or destruction (15.6.2). 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 (8.5.1.5) and the object expression refers to the complete object of x or one of that object’s base class subobjects but not x or one of its base class subobjects, the behavior is undefined. [Example:

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) { }
};

§ 15.7 266
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 (8.5.1.8) can be used during construction or destruction (15.6.2). When typeid is used in a constructor (including the mem-initializer or default member initializer (12.2) 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.

5 dynamic_casts (8.5.1.7) can be used during construction or destruction (15.6.2). When a dynamic_cast is used in a constructor (including the mem-initializer or default member initializer for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the dynamic_cast refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor or destructor’s class. If the operand of the dynamic_cast refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor’s own class or one of its bases, the dynamic_cast results in undefined behavior. [Example:

```cpp
struct V {
    virtual void f();
};

struct A : virtual V { }

struct B : virtual V {
    B(V*, A*);
};

struct D : A, B {
    D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
    typeid(*this); // type_info for B
    typeid(v); // well-defined: v has type V, a base of B yields type_info for B
    typeid(a); // undefined behavior: a not a base of B
    dynamic_cast<B*>(v); // well-defined: v of type V*, V base of B results in B*
    dynamic_cast<B*>(a); // undefined behavior, a has type A*, A not a base of B
}

—end example
```

15.8 Copying and moving class objects [class.copy]

1 A class object can be copied or moved in two ways: by initialization (15.1, 11.6), including for function argument passing (8.5.1.2) and for function value return (9.6.3); and by assignment (8.5.18). Conceptually, these two operations are implemented by a copy/move constructor (15.1) and copy/move assignment operator (16.5.3).

2 A program is ill-formed if the copy/move constructor or the copy/move assignment operator for an object is implicitly odr-used and the special member function is not accessible (Clause 14). [Note: Copying/moving one object into another using the copy/move constructor or the copy/move assignment operator does not change the layout or size of either object. —end note]

15.8.1 Copy/move constructors [class.copy ctor]

1 A non-template constructor for class X is a copy constructor if its first parameter is of type X&, const X&, volatile X& or const volatile X&, and either there are no other parameters or else all other parameters

§ 15.8.1
have default arguments (11.3.6). [Example: \texttt{X::X(const X&) and X::X(X&, int=1)} are copy constructors.]

\begin{verbatim}
    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 —
\end{verbatim}

2 A non-template constructor for class \texttt{X} is a move constructor if its first parameter is of type \texttt{X&&}, \texttt{const X&&}, \texttt{volatile X&&}, or \texttt{const volatile X&&}, and either there are no other parameters or else all other parameters have default arguments (11.3.6). [Example: \texttt{Y::Y(Y&&)} is a move constructor.]

\begin{verbatim}
    struct Y {
        Y(const Y&);
        Y(Y&&);
    }
    extern Y f(int);
    Y d(f(1)); // calls Y(Y&&)
    Y e = d; // calls Y(const Y&)

    — end example —
\end{verbatim}

3 [Note: All forms of copy/move constructor may be declared for a class. [Example:]

\begin{verbatim}
    struct X {
        X(const X&);
        X(X&); // OK
        X(X&&); // OK, but possibly not sensible
    }

    — end example — end note]
\end{verbatim}

4 [Note: If a class \texttt{X} only has a copy constructor with a parameter of type \texttt{X&}, an initializer of type \texttt{const X} or \texttt{volatile X} cannot initialize an object of type (possibly cv-qualified) \texttt{X}. [Example:]

\begin{verbatim}
    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]
\end{verbatim}

5 A declaration of a constructor for a class \texttt{X} is ill-formed if its first parameter is of type (optionally cv-qualified) \texttt{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:]

\begin{verbatim}
    struct S {
        template<typename T> S(T);
        S();
    }
    S g;

    void h() {
        S a(g); // does not instantiate the member template to produce S::S<S>(S);
        // uses the implicitly declared copy constructor
    }

    — end example —
\end{verbatim}

6 If the class definition does not explicitly declare a copy constructor, a non-explicit one is declared \textit{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 (11.4). The latter case is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor.
The implicitly-declared copy constructor for a class \( X \) will have the form
\[
X::X(const \ X&) 
\]
if each potentially constructed subobject of a class type \( M \) (or array thereof) has a copy constructor whose first parameter is of type \( \text{const } M \& \) or \( \text{const volatile } M \& \).\(^{120}\) Otherwise, the implicitly-declared copy constructor will have the form
\[
X::X(X&) 
\]
If the definition of a class \( X \) does not explicitly declare a move constructor, a non-explicit one will be implicitly declared as defaulted if and only if

\( (8.1) \) — \( X \) does not have a user-declared copy constructor,
\( (8.2) \) — \( X \) does not have a user-declared copy assignment operator,
\( (8.3) \) — \( X \) does not have a user-declared move assignment operator, and
\( (8.4) \) — \( X \) does not have a user-declared destructor.

[ \text{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
\[
X::X(X&&) 
\]
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 (11.4.3) if \( X \) has:

\( (10.1) \) — a potentially constructed subobject type \( M \) (or array thereof) that cannot be copied/moved because overload resolution (16.3), as applied to find \( M \)'s corresponding constructor, results in an ambiguity or a function that is deleted or inaccessible from the defaulted constructor,
\( (10.2) \) — a variant member whose corresponding constructor as selected by overload resolution is non-trivial,
\( (10.3) \) — any potentially constructed subobject of a type with a destructor that is deleted or inaccessible from the defaulted constructor, or,
\( (10.4) \) — for the copy constructor, a non-static data member of rvalue reference type.

A defaulted move constructor that is defined as deleted is ignored by overload resolution (16.3, 16.4). [ \text{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:

\( (11.1) \) — class \( X \) has no virtual functions (13.3) and no virtual base classes (13.1), and
\( (11.2) \) — the constructor selected to copy/move each direct base class subobject is trivial, and
\( (11.3) \) — for each non-static data member of \( X \) that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;

otherwise the copy/move constructor is \textit{non-trivial}.

A copy/move constructor that is defaulted and not defined as deleted is \textit{implicitly defined} when it is odr-used (6.2), when it is needed for constant evaluation (8.6), or when it is explicitly defaulted after its first declaration. [ \text{Note: The copy/move constructor is implicitly defined even if the implementation elided its odr-use (6.2, 15.2). — end note} \] If the implicitly-defined constructor would satisfy the requirements of a constexpr constructor (10.1.5), the implicitly-defined constructor is \textit{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. [ \text{Note: An implicitly-declared copy/move constructor has an implied exception specification (18.4). — end note} \]

The implicitly-defined copy/move constructor for a non-union class \( X \) performs a memberwise copy/move of its bases and members. [ \text{Note: Default member initializers of non-static data members are ignored. See also the example in 15.6.2. — end note} \] The order of initialization is the same as the order of initialization of bases and members in a user-defined constructor (see 15.6.2). Let \( x \) be either the parameter of the constructor

\( ^{120} \) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a \textit{volatile} lvalue; see C.1.9.
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:

(14.1) — if the member is an array, each element is direct-initialized with the corresponding subobject of x;

(14.2) — if a member m has rvalue reference type T&&, it is direct-initialized with static_cast<T&&>(x.m);

(14.3) — 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 15.6.2).

The implicitly-defined copy/move constructor for a union X copies the object representation (6.7) of X.

15.8.2 Copy/move assignment operator

A user-declared copy assignment operator X::operator= is a non-static non-template member function of
class X with exactly one parameter of type X, X&, const X&, volatile X& or const volatile X&. [Note: An
overloaded assignment operator must be declared to have only one parameter; see 16.5.3. — 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 const X
cannot be assigned to an object of type X. [Example:

```c
struct X {
    X();
    X& operator=(X&);  // error: X::operator=(X&) cannot assign cx into x
    const X cx;
    X x;
    void f() {
        x = cx;
    }
}
```
— end example] — end note]

2 If the class definition does not explicitly declare a copy assignment operator, one is declared implicitly.
If the class definition declares a move constructor or move assignment operator, the implicitly declared
copy assignment operator is defined as deleted; otherwise, it is defined as defaulted (11.4). The latter
case is deprecated if the class has a user-declared copy constructor or a user-declared destructor. The
implicitly-declared copy assignment operator for a class X will have the form

X& X::operator=(const X&) if

(2.1) — each direct base class B of X has a copy assignment operator whose parameter is of type const B&,
const volatile B& or B, and

(2.2) — 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 const M&, const volatile M& or M.

Otherwise, the implicitly-declared copy assignment operator will have the form

X& X::operator=(X&) 3

A user-declared move assignment operator X::operator= is a non-static non-template member function of
class X with exactly one parameter of type X&&, const X&&, volatile X&&, or const volatile X&&. [Note: An
overloaded assignment operator must be declared to have only one parameter; see 16.5.3. — end note]
[Note: More than one form of move assignment operator may be declared for a class. — end note]

If the definition of a class X does not explicitly declare a move assignment operator, one will be implicitly
declared as defaulted if and only if

(4.1) — X does not have a user-declared copy constructor,

(4.2) — X does not have a user-declared move constructor,

121) 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.

122) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile
lvalue; see C.1.9.
(4.3) — X does not have a user-declared copy assignment operator, and
(4.4) — 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]

5 The implicitly-declared move assignment operator for a class X will have the form

```cpp
X& X::operator=(X&&);
```

6 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.

7 A defaulted copy/move assignment operator for class X is defined as deleted if X has:

- (7.1) a variant member with a non-trivial corresponding assignment operator and X is a union-like class, or
- (7.2) a non-static data member of const non-class type (or array thereof), or
- (7.3) a non-static data member of reference type, or
- (7.4) a direct non-static data member of class type M (or array thereof) or a direct base class M that cannot be copied/moved because overload resolution (16.3), 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 (16.3, 16.4).

8 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 (16.5.3). A using-declaration (10.3.3) 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.

9 A copy/move assignment operator for class X is trivial if it is not user-provided and if:

- (9.1) class X has no virtual functions (13.3) and no virtual base classes (13.1), and
- (9.2) the assignment operator selected to copy/move each direct base class subobject is trivial, and
- (9.3) for each non-static data member of X that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial;

otherwise the copy/move assignment operator is non-trivial.

10 A copy/move assignment operator for a class X that is defaulted and not defined as deleted is implicitly defined when it is odr-used (6.2) (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 (8.6), or when it is explicitly defaulted after its first declaration. The implicitly-defined copy/move assignment operator is constexpr if

- (10.1) X is a literal type, and
- (10.2) the assignment operator selected to copy/move each direct base class subobject is a constexpr function, and

§ 15.8.2 271
for each non-static data member of \( X \) that is of class type (or array thereof), the assignment operator selected to copy/move that member is a constexpr function.

Before the defaulted copy/move assignment operator for a class is implicitly defined, all non-user-provided copy/move assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [Note: An implicitly-declared copy/move assignment operator has an implied exception specification (18.4). —end note]

The implicitly-defined copy/move assignment operator for a non-union class \( X \) performs memberwise copy-move assignment of its subobjects. The direct base classes of \( X \) are assigned first, in the order of their declaration in the base-specifier-list, and then the immediate non-static data members of \( X \) are assigned, in the order in which they were declared in the class definition. Let \( x \) be either the parameter of the function or, for the move operator, an rvalue referring to the parameter. Each subobject is assigned in the manner appropriate to its type:

- if the subobject is of class type, as if by a call to \( \text{operator=} \) with the subobject as the object expression and the corresponding subobject of \( x \) as a single function argument (as if by explicit qualification; that is, ignoring any possible virtual overriding functions in more derived classes);
- if the subobject is an array, each element is assigned, in the manner appropriate to the element type;
- if the subobject is of scalar type, the built-in assignment operator is used.

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined copy/move assignment operator. [Example:

```
struct V { }
struct A : virtual V { }
struct B : virtual V { }
struct C : B, A { }
```

It is unspecified whether the virtual base class subobject \( V \) is assigned twice by the implicitly-defined copy/move assignment operator for \( C \). —end example]

The implicitly-defined copy assignment operator for a union \( X \) copies the object representation (6.7) of \( X \).

### 15.8.3 Copy/move elision

When certain criteria are met, an implementation is allowed to omit the copy/move construction of a class object, even if the constructor selected for the copy/move operation and/or the destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy/move operation as simply two different ways of referring to the same object. If the first parameter of the selected constructor is an rvalue reference to the object’s type, the destruction of that object occurs when the target would have been destroyed; otherwise, the destruction occurs at the later of the times when the two objects would have been destroyed without the optimization. This elision of copy/move operations, called copy elision, is permitted in the following circumstances (which may be combined to eliminate multiple copies):

- in a \texttt{return} statement in a function with a class return type, when the \texttt{expression} is the name of a non-volatile automatic object (other than a function parameter or a variable introduced by the \texttt{exception-declaration} of a handler (18.3)) 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

123 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.
Copy elision is required where an expression is evaluated in a context requiring a constant expression (8.6) and in constant initialization (6.8.3.2). [Note: Copy elision might not be performed if the same expression is evaluated in another context. —end note]

2 [Example:
   ```cpp
class Thing {
   public:
      Thing();
      ~Thing();
      Thing(const Thing&);
   };

   Thing f() {
      Thing t;
      return t;
   }

   Thing t2 = f();

   struct A {
      void *p;
      constexpr A(): p(this) {};
   };

   constexpr A 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 g() {
      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]

3 In the following copy-initialization contexts, a move operation might be used instead of a copy operation:

   (3.1) — If the expression in a return statement (9.6.3) 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

   (3.2) — if the operand of a throw-expression (8.5.17) 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 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 is 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 to be called if elision is not performed, and the selected constructor 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 f(bool b) {
    Thing t;
    if (b)
        throw t;
    // OK: Thing(Thing&&) used (or elided) to throw t
    return t;
    // OK: Thing(Thing&&) used (or elided) to return t
}

Thing t2 = f(false);
// OK: no extra copy/move performed, t2 constructed by call to f

struct Weird {
    Weird();
    Weird(Weird&&);
};

Weird g() {
    Weird w;
    return w;
    // OK: first overload resolution fails, second overload resolution selects Weird(Weird&&)
}

—end example]

15.9 Comparisons

15.9.1 Defaulted comparison operator functions

A defaulted comparison operator function (8.5.8, 8.5.9, 8.5.10) for some class C shall be a non-template function declared in the member-specification of C that is

— (1.1) a non-static member of C having one parameter of type const C&, or

— (1.2) a friend of C having two parameters of type const C&.

15.9.2 Three-way comparison

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 (16.3.3.1), class member access expressions (8.5.1.5), and array subscript expressions (8.5.1.1) applied to \( x \). The type of the expression \( x_i \text{ <=> } x_i \) is denoted by \( R_i \). It is unspecified whether virtual base class subobjects are compared more than once.

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 \( R_0, R_1, \ldots, R_{n-1} \). [Note: Otherwise, the program will be ill-formed if the expression \( x_i \text{ <=> } x_i \) 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 type \( R \) of the defaulted three-way comparison operator function with parameters \( x \) and \( y \) of the same type is determined by comparing corresponding elements \( x_i \) and \( y_i \) in the expanded lists of subobjects for \( x \) and \( y \) until the first index \( i \) where \( x_i \text{ <=> } y_i \) 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 \( T_0, T_1, \ldots, T_{n-1} \) is defined as follows:

— (4.1) If any \( T_i \) is not a comparison category type (21.10.2), \( U \) is void.

— (4.2) 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 \( T_j \) is std::partial_ordering or std::weak_ordering, \( U \) is std::weak_equality (21.10.2.2).  

— (4.3) Otherwise, if at least one \( T_i \) is std::strong_equality, \( U \) is std::strong_equality (21.10.2.3).

— (4.4) Otherwise, if at least one \( T_i \) is std::partial_ordering, \( U \) is std::partial_ordering (21.10.2.4).

§ 15.9.2
— Otherwise, if at least one $T_i$ is `std::weak_ordering`, $U$ is `std::weak_ordering` (21.10.2.5).

— Otherwise, $U$ is `std::strong_ordering` (21.10.2.6). [Note: In particular, this is the result when $n$ is 0. —end note]

15.9.3 Other comparison operators  [class.rel.eq]

1 A defaulted relational (8.5.9) or equality (8.5.10) 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

1. overload resolution (16.3), as applied to $x <=> y$ (also considering synthesized candidates with reversed order of parameters (16.3.1.2)), results in an ambiguity or a function that is deleted or inaccessible from the operator function, or

2. the operator `@` cannot be applied to the return type of $x <=> y$ or $y <=> x$.

Otherwise, the operator function yields $x <=> y$ if an `operator<=>` with the original order of parameters was selected, or $0 @ y <=> x$ otherwise.

3 [Example:

```cpp
struct C {
  friend std::strong_equality operator<=>(const C&, const C&);
  friend bool operator==(const C& x, const C& y) = default;  // OK, returns x <=> y == 0
  bool operator<(const C&) = default;  // OK, function is deleted
};

— end example]

§ 15.9.3
16 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 an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called overload resolution and is defined in 16.3. [Example:

```c++
double abs(double);
int abs(int);
abs(1); // calls abs(int);
abs(1.0); // calls abs(double);
— end example]
```

16.1 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 (10.3.3). 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 (18.4), or both cannot be overloaded.

(2.2) — Member function declarations with the same name and the same parameter-type-list (11.3.5) cannot be overloaded if any of them is a static member function declaration (12.2.3). 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 (16.3.1) 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();
    void f(); // ill-formed
    void f() const; // ill-formed
    void f() const volatile; // ill-formed
    void g();
    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 (11.3.5) 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 (11.3.5). [Example:]

```c++
class Y {
    void h() &;
    void h() const &; // OK
    void h() &&; // OK, all declarations have a ref-qualifier
};
```
void i() &;  // ill-formed, prior declaration of i
void i() const;  // has a ref-qualifier

— end example

3 [ Note: As specified in 11.3.5, function declarations that have equivalent parameter declarations and requires-clauses, if any (17.4.2), 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 (10.1.3). [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:
enum E { a };
void f(int i) { /* ... */ }
void f(E i) { /* ... */ }
— end example]

(3.2) — Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration (11.3.5). Only the second and subsequent array dimensions are significant in parameter types (11.3.4). [Example:
int f(char[]);
int f(char[]);  // same as f(char*)
int f(char[7]);  // same as f(char*)
int f(char[9]);  // same as f(char*)

int g(char(*)[10]);
int g(char[5][10]);  // same as g(char(*)[10])
int g(char[7][10]);  // same as g(char(*)[10])
int g(char(*)[20]);  // different from g(char(*)[10])
— end example]

(3.3) — Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (11.3.5). [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 (int);  // redeclaration of f(int)
int f (const 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`”.

— Two parameter declarations that differ only in their default arguments are equivalent. [Example: Consider the following:

```c
void f (int i, int j);
void f (int i, int j = 99); // OK: redeclaration of f(int, int)
void f (int i = 88, int j); // OK: redeclaration of f(int, int)
void f (); // OK: overloaded declaration of f

void prog () {
    f (1, 2); // OK: call f(int, int)
    f (1); // OK: call f(int, int)
    f (); // error: f(int, int) or f() ?
}
— end example]
— end note]

16.2 Declaration matching

Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (16.1) and equivalent trailing requires-clauses, if any (Clause 11). 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.

void h(D* pd) {
    pd->f(1); // error:
    // D::f(const char*) hides B::f(int)
    pd->B::f(1); // OK
    pd->f("Ben"); // OK, calls D::f
}
— end example]

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
    }
}
```

124) 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]

16.3 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 (16.3.1.1.1);

(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 (16.3.1.1.2);

(2.3) — invocation of the operator referenced in an expression (16.3.1.2);

(2.4) — invocation of a constructor for default- or direct-initialization (11.6) of a class object (16.3.1.3);

(2.5) — invocation of a user-defined conversion for copy-initialization (11.6) of a class object (16.3.1.4);

(2.6) — invocation of a conversion function for initialization of an object of a non-class type from an expression of class type (16.3.1.5); and

(2.7) — invocation of a conversion function for conversion to a glvalue or class prvalue to which a reference (11.6.3) will be directly bound (16.3.1.6).

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 (16.3.2).

(2.9) — Then the best viable function is selected based on the implicit conversion sequences (16.3.3.1) 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 (Clause 14) in the context in which it is used, the program is ill-formed.

16.3.1 Candidate functions and argument lists [over.match.funcs]

1 The subclauses of 16.3.1 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 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* to denote the object to be operated on. Since arguments and parameters are associated by position within their respective lists, the convention is that the implicit object parameter, if present, is always the first parameter and the implied object argument, if present, is always the first argument.

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 (16.3.3). — 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 (16.3.3.2). — 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 (16.3.3, 16.3.3.1). [Example:

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 (17.9.3, 17.9.2). 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 (15.8) that is defined as deleted is excluded from the set of candidate functions in all contexts.

125) 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 (11.3.5) are treated equivalently for the remainder of overload resolution.
16.3.1.1 Function call syntax

1 In a function call (8.5.1.2)

\[
p\text{-expression} (\text{expression-list}_{\text{opt}})
\]

if the \textit{postfix-expression} denotes a set of overloaded functions and/or function templates, overload resolution is applied as specified in 16.3.1.1.1. If the \textit{postfix-expression} denotes an object of class type, overload resolution is applied as specified in 16.3.1.1.2.

2 If the \textit{postfix-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. [\textit{Note}: The resolution of the address of an overload set in other contexts is described in 16.4. — end note]

16.3.1.1.1 Call to named function

1 Of interest in 16.3.1.1.1 are only those function calls in which the \textit{postfix-expression} ultimately contains a name that denotes one or more functions that might be called. Such a \textit{postfix-expression}, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

\[
\begin{align*}
\textit{postfix-expression}: & \\
& \textit{postfix-expression} . \text{id-expression} \\
& \textit{postfix-expression} \rightarrow \text{id-expression} \\
& \text{primary-expression}
\end{align*}
\]

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{\rightarrow} operator. Since the construct \texttt{A->B} is generally equivalent to \texttt{(*A).B}, the rest of Clause 16 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 16 assumes that the \textit{postfix-expression} that is the left operand of the \texttt{.} operator has type “\texttt{cv T}” where \texttt{T} denotes a class.\textsuperscript{126} 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 (13.2). 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 (16.3.1).

3 In unqualified function calls, the name is not qualified by an \texttt{->} or \texttt{\rightarrow} 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 function calls (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 (1) entirely of non-member functions or (2) entirely of member functions of some class \texttt{T}. In case (1), the argument list is the same as the \textit{expression-list} in the call. In case (2), the argument list is the \textit{expression-list} in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword \texttt{this} (12.2.2.1) is in scope and refers to class \texttt{T}, or a derived class of \texttt{T}, then the implied object argument is \texttt{(*this)}. If the keyword \texttt{this} is not in scope or refers to another class, then a contrived object of type \texttt{T} becomes the implied object argument.\textsuperscript{127} If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of \texttt{T}, the call is ill-formed.

16.3.1.1.2 Call to object of class type

1 If the \textit{postfix-expression} \texttt{E} in the function call syntax evaluates to a class object of type “\texttt{cv T}”, then the set of candidate functions includes at least the function call operators of \texttt{T}. The function call operators of \texttt{T} are obtained by ordinary lookup of the name \texttt{operator()} in the context of \texttt{(E).operator()}.

2 In addition, for each non-explicit conversion function declared in \texttt{T} of the form

\[
\texttt{operator conversion-type-id () cv-qualifier ref-qualifier}_{\text{opt}} \text{noexcept-specifier}_{\text{opt}} \text{attribute-specifier-seq}_{\text{opt}};
\]

where \texttt{cv-qualifier} is the same cv-qualification as, or a greater cv-qualification than, \texttt{cv}, and where \textit{conversion-type-id} denotes the type “pointer to function of (\texttt{P}_1, \ldots, \texttt{P}_n) returning \texttt{R}”, or the type “reference to pointer...

\textsuperscript{126} Note that cv-qualifiers on the type of objects are significant in overload resolution for both glvalue and class prvalue objects.

\textsuperscript{127} An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.
to function of $(P_1, \ldots, P_n)$ returning $R$, or the type “reference to function of $(P_1, \ldots, P_n)$ returning $R$”, a surrogate call function with the unique name call-function and having the form

$$R \; \text{call-function}(\text{conversion-type-id} \; F, \; P_1 \; a_1, \ldots, \; P_n \; a_n) \{ \text{return} \; F(a_1, \ldots, a_n); \}$$

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of $T$ provided the function is not hidden within $T$ by another intervening declaration.\(^{128}\)

3 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.

4 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument $(E)$. \(^{[\text{Note: When comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. — end note]}\)

---

### 16.3.1.2 Operators in expressions \([\text{over.match.oper]}\)

1 If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to 8.5. \(^{[\text{Note: Because . , . , and :: cannot be overloaded, these operators are always built-in operators interpreted according to 8.5. ?: cannot be overloaded, but the rules in this subclause are used to determine the conversions to be applied to the second and third operands when they have class or enumeration type (8.5.16). — end note]}\)

---

\(^{128}\) Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.
denotes one of the operators covered in the specified subclause). However, the operands are sequenced in the order prescribed for the built-in operator (8.5).

Table 12 — 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>16.5.1</td>
<td>@a</td>
<td>(a).operator@()</td>
<td>operator@(a)</td>
</tr>
<tr>
<td>16.5.2</td>
<td>a@b</td>
<td>(a).operator@ (b)</td>
<td>operator@ (a, b)</td>
</tr>
<tr>
<td>16.5.3</td>
<td>a=b</td>
<td>(a).operator= (b)</td>
<td></td>
</tr>
<tr>
<td>16.5.5</td>
<td>a[b]</td>
<td>(a).operator[] (b)</td>
<td></td>
</tr>
<tr>
<td>16.5.6</td>
<td>a-&gt;</td>
<td>(a).operator-&gt;()</td>
<td></td>
</tr>
<tr>
<td>16.5.7</td>
<td>a@</td>
<td>(a).operator@ (0)</td>
<td>operator@ (a, 0)</td>
</tr>
</tbody>
</table>

3 For a unary operator @ with an operand of 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, three sets of candidate functions, designated member candidates, non-member candidates and built-in 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@ (16.3.1.1.1); 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 16.6 that, compared to the given operator,

(3.3.1) — have the same operator name, and

(3.3.2) — accept the same number of operands, and

(3.3.3) — accept operand types to which the given operand or operands can be converted according to 16.3.3.1, and

(3.3.4) — do not have the same parameter-type-list as any non-member candidate that is not a function template specialization.

4 For the built-in assignment operators, conversions of the left operand are restricted as follows:

(4.1) — no temporaries are introduced to hold the left operand, and

(4.2) — no user-defined conversions are applied to the left operand to achieve a type match with the left-most parameter of a built-in candidate.

5 For all other operators, no such restrictions apply.

6 The set of candidate functions for overload resolution for some operator @ is the union of the member candidates, the non-member candidates, and the built-in candidates for that operator @. If that operator is a relational (8.5.9) or equality (8.5.10) operator with operands x and y, then for each member, non-member, or built-in candidate for the operator <=>

(6.1) — that operator is added to the set of candidate functions for overload resolution if x <= y @ 0 is well-formed using that operator<=>; and

(6.2) — a synthesized candidate is added to the candidate set where the order of the two parameters is reversed if 0 @ y <= x is well-formed using that operator<=>; where in each case operator<=> candidates are not considered for the recursive lookup of operator @.

7 The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 16.3.2 and 16.3.3.129 [Example:

129] If the set of candidate functions is empty, overload resolution is unsuccessful.
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 an operator\(\lll\) candidate is selected by overload resolution for an operator \(\l\), but \(\l\) is not \(\lll\), \(x \l y\) is interpreted as \(0 \l y \lll x\) if the selected candidate is a synthesized candidate with reversed order of parameters, or \(x \lll y \l 0\) otherwise, using the selected operator\(\lll\) 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 (16.3.3.1.2) is not applied. Then the operator is treated as the corresponding built-in operator and interpreted according to 8.5. [Example:

```cpp
struct X {
    operator double();
};
struct Y {
    operator int*();
};
int *a = Y() + 100.0; // error: pointer arithmetic requires integral operand
int *b = Y() + X(); // error: pointer arithmetic requires integral operand
—end example
```

The second operand of operator \(\l\) is ignored in selecting an operator\(\l\) function, and is not an argument when the operator\(\l\) function is called. When operator\(\l\) returns, the operator \(\l\) is applied to the value returned, with the original second operand.\(^{130}\)

If the operator is the operator \(.,\) the unary operator \&, or the operator \(\l\), and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to 8.5.

[Note: The lookup rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:

```cpp
struct A {
};
void operator + (A, A);
struct B {
    void operator + (B);
    void f ();
};
A a;
void B::f() {
    operator+ (a,a); // error: global operator hidden by member
    a + a; // OK: calls global operator+
}
—end note]

16.3.1.3 Initialization by constructor

When objects of class type are direct-initialized (11.6), copy-initialized from an expression of the same or a derived class type (11.6), or default-initialized (11.6), 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, the
candidate functions are all the converting constructors (15.3.1) of that class. The argument list is the expression-list or assignment-expression of the initializer.

16.3.1.4 Copy-initialization of class by user-defined conversion

1 Under the conditions specified in 11.6, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized. Overload resolution is used to select the user-defined conversion to be invoked. [ Note: The conversion performed for indirect binding to a reference to a possibly cv-qualified class type is determined in terms of a corresponding non-reference copy-initialization. — end note ] Assuming that “cv1 T” is the type of the object being initialized, with T a class type, the candidate functions are selected as follows:

(1.1) — The converting constructors (15.3.1) of T are candidate functions.

(1.2) — When the type of the initializer expression is a class type “cv S”, the non-explicit conversion functions of S and its base classes are considered. When initializing a temporary object (12.2) to be bound to the first parameter of a constructor where the parameter is of type “reference to possibly cv-qualified T” and the constructor is called with a single argument in the context of direct-initialization of an object of type “cv2 T”, explicit conversion functions are also considered. Those that are not hidden within S and yield a type whose cv-unqualified version is the same type as T or is a derived class thereof are candidate functions. Conversion functions that return “reference to X” return lvalues or xvalues, depending on the type of reference, of type X and are therefore considered to yield X for this process of selecting candidate functions.

2 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 ]

16.3.1.5 Initialization by conversion function

1 Under the conditions specified in 11.6, 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 (16.3.3.1.1) 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 (7.5) 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. Conversion functions that return “reference to cv2 X” return lvalues or xvalues, depending on the type of reference, of type “cv2 X” and are therefore considered to yield X for this process of selecting candidate functions.

2 The argument list has one argument, which is the initializer expression. [ Note: This argument will be compared against the implicit object parameter of the conversion functions. — end note ]

16.3.1.6 Initialization by conversion function for direct reference binding

1 Under the conditions specified in 11.6.3, a reference can be bound directly to a gvalue or class prvalue that is 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 “value reference to cv2 T2” (when initializing an lvalue reference or an rvalue reference to function) or “cv2 T2” or “rvalue reference to cv2 T2” (when initializing an rvalue reference or an rvalue reference to function), where “cv1 T” is reference-compatible (11.6.3) with “cv2 T2”, are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within S and yield type “lvalue reference to cv2 T2” or “cv2 T2” or “rvalue reference to cv2 T2”, respectively, where T2 is the same type as T or can be converted to type T with a qualification conversion (7.5), 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]

16.3.1.7 Initialization by list-initialization

When objects of non-aggregate class type \( T \) are list-initialized such that 11.6.4 specifies that overload resolution is performed according to the rules in this subclause, overload resolution selects the constructor in two phases:

1. Initially, the candidate functions are the initializer-list constructors (11.6.4) of the class \( T \) and the argument list consists of the initializer list as a single argument.

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 (16.3.1.3, 16.3.1.4), 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]

16.3.1.8 Class template argument deduction

When resolving a placeholder for a deduced class type (10.1.7.5) where the template-name names a primary class template \( C \), a set of functions and function templates is formed comprising:

1. If \( C \) is defined, for each constructor of \( C \), a function template with the following properties:
   1.1. The template parameters are the template parameters of \( C \) followed by the template parameters (including default template arguments) of the constructor, if any.
   1.2. The types of the function parameters are those of the constructor.
   1.3. The return type is the class template specialization designated by \( C \) and template arguments corresponding to the template parameters of \( C \).

2. If \( C \) is not defined or does not declare any constructors, an additional function template derived as above from a hypothetical constructor \( C() \).

3. An additional function template derived as above from a hypothetical constructor \( C(C) \), called the copy deduction candidate.

4. For each deduction-guide, a function or function template with the following properties:
   1.4.1. The template parameters, if any, and function parameters are those of the deduction-guide.
   1.4.2. The return type is the simple-template-id of the deduction-guide.

Initialization and overload resolution are performed as described in 11.6 and 16.3.1.3, 16.3.1.4, or 16.3.1.7 (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 16.3.1.7 (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 \( C \). Each such notional constructor is considered to be explicit if the function or function template was generated from a constructor or deduction-guide that was declared explicit. 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& x, ...) noexcept; // #1
    A(T&& x, ...); // #2
};

int i;
A a1 = { i, i }; // error: explicit constructor #1 selected in copy-list-initialization during deduction,
// cannot deduce from non-forwarding rvalue reference in #2

A a2{i, i}; // OK, #1 deduces to A<int> and also initializes
A a3{0, i}; // OK, #2 deduces to A<int> and also initializes
```}

§ 16.3.1.8
A \( a4 = \{0, i\}; \) // OK, \#2 deduces to \(<\text{int}>\) and also initializes

```cpp
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 \(<\text{int}>\) and \#2 initializes
A \( a7 = \{0, 1\}; \) // error: \#3 deduces to \(<\text{int}>\&\), \#1 and \#2 declare same constructor
A \( a8(0,i); \) // error: \#3 deduces to \(<\text{int}>\&\), \#1 and \#2 declare same constructor

```cpp
template <class T> struct B {
    template <class U> using TA = T;
    template <class U> B(U, TA<U>);
};
```

B \( b((\text{int}*)0, (\text{char}*)0); \) // OK, deduces \(<\text{char}>\)

--- end example ---

### 16.3.2 Viable functions

From the set of candidate functions constructed for a given context (16.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences and associated constraints (17.4.2) for the best fit (16.3.3). 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.

1. If there are \( m \) arguments in the list, all candidate functions having exactly \( m \) parameters are viable.
2. A candidate function having fewer than \( m \) parameters is viable only if it has an ellipsis in its parameter list (11.3.5). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to “match the ellipsis” (16.3.3.1.3).
3. A candidate function having more than \( m \) parameters is viable only if the \( (m+1) \)-st parameter has a default argument (11.3.6). 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, those constraints shall be satisfied (17.4.2).

Third, for \( F \) to be a viable function, there shall exist for each argument an implicit conversion sequence (16.3.3.1) 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 16.3.3.1.4).

### 16.3.3 Best viable function

Define \( \text{ICS}_i(F) \) as follows:

1. If \( F \) is a static member function, \( \text{ICS}_i(F) \) is defined such that \( \text{ICS}_i(F) \) is neither better nor worse than \( \text{ICS}_i(G) \) for any function \( G \), and, symmetrically, \( \text{ICS}_i(G) \) is neither better nor worse than \( \text{ICS}_i(F) \); otherwise,
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 \( F \). 16.3.3.1 defines the implicit conversion sequences and 16.3.3.2 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 \( F1 \) is defined to be a \textit{better} function than another viable function \( F2 \) if for all arguments \( i \), \( \text{ICS}_i(F1) \) is not a worse conversion sequence than \( \text{ICS}_i(F2) \), and then

--- end example ---
— the context is an initialization by user-defined conversion (see 11.6, 16.3.1.5, and 16.3.1.6) 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 [\textit{Example}:

```cpp
struct A {
    A();
    operator int();
    operator double();
} a;
int i = a; // a.operator int() followed by no conversion is better than
          // a.operator double() followed by a conversion to int
float x = a; // ambiguous: both possibilities require conversions,
              // and neither is better than the other
end example]

— end example ] or, if not that,

— the context is an initialization by conversion function for direct reference binding (16.3.1.6) 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 [\textit{Example}:

```cpp
template <class T> struct A {
    operator T&(); // #1
    operator T&&(); // #2
};
typedef int Fn();
A<Fn> a;
Fn& lf = a; // calls #1
Fn&& rf = a; // calls #2
end example]

— end example] or, if not that,

— \( F_1 \) is not a function template specialization and \( F_2 \) is a function template specialization, or, if not that,

— \( F_1 \) and \( F_2 \) are function template specializations, and the function template for \( F_1 \) is more specialized than the template for \( F_2 \) according to the partial ordering rules described in 17.6.6.2, or, if not that,

— \( F_1 \) and \( F_2 \) are non-template functions with the same parameter-type-lists, and \( F_1 \) is more constrained than \( F_2 \) according to the partial ordering of constraints described in 17.4.4, or if not that,

— \( F_1 \) is a constructor for a class \( D \), \( F_2 \) is a constructor for a base class \( B \) of \( D \), and for all arguments the corresponding parameters of \( F_1 \) and \( F_2 \) have the same type. [\textit{Example}:

```cpp
struct A {
    A(int = 0);
};

struct B: A {
    using A::A;
    B();
};

int main() {
    B b; // OK, B::B()
}
end example] or, if not that,

— \( F_1 \) is an operator function for a relational (8.5.9) or equality (8.5.10) operator and \( F_2 \) is an operator function for a three-way comparison operator (8.5.8) [\textit{Example}:

```cpp
struct S {
    auto operator<=>(const S&, const S&) = default; // #1
    bool operator<(const S&, const S&); // #2
};
bool b = S() < S(); // calls #2
end example] or, if not that,

— \( F_1 \) and \( F_2 \) are operator functions for \texttt{operator<=>} and \( F_2 \) is a synthesized candidate with reversed order of parameters and \( F_1 \) is not [\textit{Example}:
struct S {
  std::weak_ordering operator<=>(const S&, int); // #1
  std::weak_ordering operator<=>(int, const S&); // #2
};
bool b = 1 < S(); // calls #2
— end example

— F1 is generated from a deduction-guide (16.3.1.8) and F2 is not, or, if not that,
— F1 is the copy deduction candidate (16.3.1.8) and F2 is not, or, if not that,
— F1 is generated from a non-template constructor and F2 is generated from a constructor template.

[Example:

```cpp
template <class T> struct A {
  using value_type = T;
  A(value_type); // #1
  A(const A&); // #2
  A(T, T, int); // #3
  template<class U>
  A(int, T, U); // #4
  // #5 is the copy deduction candidate, A(A)
};

A x(1, 2, 3); // uses #3, generated from a non-template constructor

template <class T>
A(T) -> A<T>; // #6, less specialized than #5

A a(42); // uses #6 to deduce A<int> and #1 to initialize
A b = a; // uses #5 to deduce A<int> and #2 to initialize

template <class T>
A(A<T>) -> A<A<T>>; // #7, as specialized as #5

A b2 = a; // uses #7 to deduce A<A<int>> and #1 to initialize
— end example
```

2 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.133 [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
```

133) 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, \( W \) is either 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.
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]

16.3.3.1 Implicit conversion sequences

1 An implicit conversion sequence is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is an implicit conversion as defined in Clause 7, which means it is governed by the rules for initialization of an object or reference by a single expression (11.6, 11.6.3).

2 Implicit conversion sequences are concerned only with the type, cv-qualification, and value category of the argument and how these are converted to match the corresponding properties of the parameter. 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 (11.4.3), 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.

3 A well-formed implicit conversion sequence is one of the following forms:

(3.1) — a standard conversion sequence (16.3.3.1.1),
(3.2) — a user-defined conversion sequence (16.3.3.1.2), or
(3.3) — an ellipsis conversion sequence (16.3.3.1.3).

4 However, if the target is

4.1 — the first parameter of a constructor or
4.2 — the implicit object parameter of a user-defined conversion function

and the constructor or user-defined conversion function is a candidate by

4.3 — 16.3.1.3, when the argument is the temporary in the second step of a class copy-initialization,
4.4 — 16.3.1.4, 16.3.1.5, or 16.3.1.6 (in all cases), or
4.5 — the second phase of 16.3.1.7 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:

```cpp
struct Y { Y(int); };
struct A { operator int(); };
Y y1 = A();          // error: A::operator int() is not a candidate

struct X { };   
struct B { operator X(); };  
B b;   
X x(b);          // error: B::operator X() is not a candidate
```
For the case where the parameter type is a reference, see 16.3.3.1.4.

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 16; 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 \texttt{A} can be initialized from an argument of type \texttt{const A}. The implicit conversion sequence for that case is the identity sequence; it contains no “conversion” from \texttt{const A} to \texttt{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 (16.3.3.1.1).

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 (16.3.3.1.1).

If no sequence of conversions can be found to convert an argument to a parameter type, an implicit conversion sequence cannot be formed.

If several different sequences of conversions exist that each convert 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 16.3.3.2, 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:

\begin{verbatim}
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 → C (via constructor)  
// and an (ambiguous) conversion b → A (via constructor or conversion function)
void f(B) { }  
f(b); // OK, unambiguous  
\end{verbatim}

— 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.

\section*{16.3.3.1.1 Standard conversion sequences} [over.ics.scs]

Table 13 summarizes the conversions defined in Clause 7 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. [Note: These categories are orthogonal with respect to value category, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the value category or data representation of the type; and the Promotions and Conversions do not change the value category or cv-qualification of the type. — end note]

[Note: As described in Clause 7, a standard conversion sequence is either the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the other four categories. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: \texttt{Lvalue Transformation, Promotion or Conversion, Qualification Adjustment}. — end note]
Each conversion in Table 13 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (16.3.3.2). 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 (16.3.3.1.4). 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.

<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></td>
</tr>
<tr>
<td>Lvalue-to-rvalue conversion</td>
<td>Lvalue Transformation</td>
<td>Exact Match</td>
<td>7.1</td>
</tr>
<tr>
<td>Array-to-pointer conversion</td>
<td></td>
<td></td>
<td>7.2</td>
</tr>
<tr>
<td>Function-to-pointer conversion</td>
<td></td>
<td></td>
<td>7.3</td>
</tr>
<tr>
<td>Qualification conversions</td>
<td>Qualification Adjustment</td>
<td></td>
<td>7.5</td>
</tr>
<tr>
<td>Function pointer conversion</td>
<td></td>
<td></td>
<td>7.13</td>
</tr>
<tr>
<td>Integral promotions</td>
<td>Promotion</td>
<td>Promotion</td>
<td>7.6</td>
</tr>
<tr>
<td>Floating-point promotion</td>
<td></td>
<td></td>
<td>7.7</td>
</tr>
<tr>
<td>Integral conversions</td>
<td></td>
<td></td>
<td>7.8</td>
</tr>
<tr>
<td>Floating-point conversions</td>
<td></td>
<td></td>
<td>7.9</td>
</tr>
<tr>
<td>Floating-integral conversions</td>
<td></td>
<td></td>
<td>7.10</td>
</tr>
<tr>
<td>Pointer conversions</td>
<td></td>
<td></td>
<td>7.11</td>
</tr>
<tr>
<td>Pointer-to-member conversions</td>
<td></td>
<td></td>
<td>7.12</td>
</tr>
<tr>
<td>Boolean conversions</td>
<td></td>
<td></td>
<td>7.14</td>
</tr>
</tbody>
</table>

16.3.3.1.2 User-defined conversion sequences

A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (15.3) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (15.3.1), 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 (15.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

2 The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. 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 16.3.3 and 16.3.3.1).

3 If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

4 A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a constructor (i.e., a user-defined conversion function) is called for those cases.

16.3.3.1.3 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 8.5.1.2).

16.3.3.1.4 Reference binding

When a parameter of reference type binds directly (11.6.3) 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 (16.3.3.1). [Example:

```c
struct A {};
struct B : public A {} b;
int f(A&);  // A
t int f(B&);  // B
int i = f(b); // calls f(B&), an exact match, rather than f(A&), a conversion
```
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 (16.3.3.1.2), 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 16.3.3.1. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the referenced type and does not constitute a conversion.

Except for an implicit object parameter, for which see 16.3.1, a standard 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 11.6.3). —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 a standard conversion sequence, however. [Example: A function with an “lvalue reference to int” parameter can be a viable candidate even if the corresponding argument is an int bit-field. The formation of implicit conversion sequences treats the int bit-field as an int lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-const lvalue reference to a bit-field (11.6.3). —end example]

When an argument is an initializer list (11.6.4), 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 (11.6.1), 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 (11.6.4). —end example]

Otherwise, if the parameter type is an aggregate class X and the initializer list has a single element of type cv U, where U is X or a class derived from X, the implicit conversion sequence is the one required to convert the element to the parameter type.

Otherwise, if the parameter type is a character array and the initializer list has a single element that is an appropriately-typed string literal (11.6.2), the implicit conversion sequence is the identity conversion.

Otherwise, if the parameter type is std::initializer_list<X> and all the elements of the initializer list can be implicitly converted to X, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to X, or if the initializer list has no elements, the identity conversion. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [Example:

```cpp
struct A { int x, y; };  
struct B { int y, x; };  
void f(A a, int);  // #1 
void f(B b, ...);  // #2 
void g(A a);  // #3 
void g(B b);  // #4 
void h() { 
  f( {.x = 1, .y = 2}, 0);  // OK; calls #1 
  f( {.y = 2, .x = 1}, 0);  // error: selects #1, initialization of a fails 
  g( {.x = 1, .y = 2} );  // due to non-matching member order (11.6.4) 
  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 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.

Since there are no parameters of array type, this will only occur as the referenced type of a reference parameter.
void f(std::initializer_list<int>);

f( {} );  // OK: f(std::initializer_list<int>) identity conversion
f( {1,2,3} );  // OK: f(std::initializer_list<int>) identity conversion
f( {'a','b'} );  // OK: f(std::initializer_list<int>) integral promotion
f( {1.0} );  // error: narrowing

struct A {
    A(std::initializer_list<double>);  // #1
    A(std::initializer_list<complex<double>>);  // #2
    A(std::initializer_list<std::string>);  // #3
};
A a{ 1.0,2.0 };  // OK, uses #1

void g(A);
g( { "foo", "bar" } );  // OK, uses #3

typedef int IA[3];
void h(const IA&);
h( { 1, 2, 3 } );  // OK: identity conversion
— end example]

6 Otherwise, if the parameter type is “array of N X”, 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.

7 Otherwise, if the parameter is a non-aggregate class X and overload resolution per 16.3.1.7 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 16.3.3.1. [Example:

```cpp
struct A {
    A(std::initializer_list<int>);
};
void f(A);
f( {a', b'} );  // OK: f(A(std::initializer_list<int>)) user-defined conversion

struct B {
    B(int, double);
};
void g(B);
g( {a', b'} );  // OK: g(B(int, double)) user-defined conversion
g( {1.0, 1.0} );  // error: narrowing

void f(B);
f( {a', b'} );  // error: ambiguous f(A) or f(B)

struct C {
    C(std::string);
};
void h(C);
h( {foo} );  // OK: h(C(std::string("foo")))

struct D {
    D(A, C);
};
void i(D);
i( {1,2}, {"bar"} );  // OK: i(D(A(std::initializer_list<int>{1,2}), C(std::string("bar"))))
```
Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (11.6.1), 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 16.3.3.1.4. [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]

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:

```c
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:

```c
void f(int);
f( {} );           // OK: identity conversion
```
— end example]

In all cases other than those enumerated above, no conversion is possible.

16.3.3.2 Ranking implicit conversion sequences [over.ics.rank]

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.

When comparing the basic forms of implicit conversion sequences (as defined in 16.3.3.1)

2.1 — a standard conversion sequence (16.3.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

2.2 — a user-defined conversion sequence (16.3.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (16.3.3.1.3).

Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:
List-initialization sequence L1 is a better conversion sequence than list-initialization sequence L2 if

- L1 converts to `std::initializer_list<X>` for some X and L2 does not, or, if not that,
- 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:

```cpp
do f1(int); // #1
do f1(std::initializer_list<long>); // #2
g1() { f1(42); } // chooses #2

void f2(std::pair<const char*, const char*>); // #3
void f2(std::initializer_list<std::string>); // #4
void g2() { f2(“foo”, “bar”); } // chooses #4
— end example]

Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

- S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form defined by 16.3.3.1.1, excluding any Lvalue Transformation; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence) or, if not that,
- 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,
- S1 and S2 are reference bindings (11.6.3) 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();
g(const int&);
g(const int&&);
int j = g(i); // calls g(const int&)
int k = g(f1()); // calls g(const int&&)
int l = g(f2()); // calls g(const int&&)

struct A {
    A& operator<<(int);
    void p() &;
    void p() &&;
};
A& operator<<(A&&, char);
A() << 1; // calls A::operator<<(int)
A() << 'c'; // calls A::operator<<(A&&, char)
A a;
a << 1; // calls A::operator<<(int)
a << 'c'; // calls A::operator<<(int)
A().p(); // calls A::p()&
a.p(); // calls A::p()&
— end example] or, if not that,

- S1 and S2 are reference bindings (11.6.3) 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,

- S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (7.5), 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 *);  // calls 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 are reference bindings (11.6.3), and the types to which the references refer are the same
type except for top-level cv-qualifiers, and the type to which the reference initialized by S2 refers
is more cv-qualified than the type to which the reference initialized by S1 refers. [Example:

int f(const int &);  // calls f(const int &)
int f(int &);        // calls f(int &)
int g(const int &);  // calls f(const int &)
int g(int);          // calls f(int &)

int i;
int j = f(i);        // calls f(int &)
int k = g(i);        // calls f(const int &)

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, a pointer to member, or std::nullptr_t 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 16.3.3); in all other contexts, the source types will be the same and the target types will be different. —end note]

16.4 Address of overloaded function

1 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 (after possibly applying the function pointer conversion (7.13)) 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

(1.1) an object or reference being initialized (11.6, 11.6.3, 11.6.4),
(1.2) the left side of an assignment (8.5.18),
(1.3) a parameter of a function (8.5.1.2),
(1.4) a parameter of a user-defined operator (16.5),
(1.5) the return value of a function, operator function, or conversion (9.6.3),
(1.6) an explicit type conversion (8.5.1.3, 8.5.1.9, 8.5.3), or
(1.7) a non-type template-parameter (17.3.2).

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 (8.4). —end note]

2 If the name is a function template, template argument deduction is done (17.9.2.2), 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 17.9.1, 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]

3 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 8.5.2.1.

4 All functions with associated constraints that are not satisfied (17.4.2) 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 F0 is eliminated if the set contains a second non-template function that is more constrained than F0 according to the partial ordering rules of 17.4.4. Any given function template specialization F1 is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of F1 according to the partial ordering rules of 17.6.6.2. After such eliminations, if any, there shall remain exactly one selected function.

[Example: int f(double);]
The initialization of \texttt{pfe} is ill-formed because no \texttt{f()} with type \texttt{int(...)\ldots} has been declared, and not because of any ambiguity. For another example,

```c
struct X {
    int f(int);
    static int f(long);
};

int (X::*p1)(int) = &X::f; // OK
int (*p2)(int) = &X::f; // error: mismatch
int (X::*p3)(long) = &X::f; // OK
int (X::*p4)(long) = &X::f; // error: wrong syntax for
                           // pointer to member
int (*p5)(long) = &(X::f); // OK
```

--- end example

[Note: If \texttt{f()} and \texttt{g()} are both overloaded functions, the cross product of possibilities must be considered to resolve \texttt{f(&g)}, or the equivalent expression \texttt{f(g)}. — end note]

[Note: Even if \texttt{B} is a public base of \texttt{D}, we have]

```c
void g(D*);
void (*p2)(B*) = &g; // error
```

--- end note

### 16.5 Overloaded operators [over.oper]

A function declaration having one of the following \textit{operator-function-ids} as its name declares an \textit{operator function}. A function template declaration having one of the following \textit{operator-function-ids} as its name declares an \textit{operator function template}. A specialization of an operator function template is also an operator function. An operator function is said to \textit{implement} the operator named in its \textit{operator-function-id}.

```c
operator function template
operator: one of
new delete new[] delete[] ( ) [ ] -> ->* 
! + - * / % ^ & |
= += -= *= /= %= ^= &= |=
== != < > <= >= <=> && ||
<< >>= <<= >>= ++ -- ,
```

[Note: The last two operators are function call (8.5.1.2) and subscripting (8.5.1.1). The operators \texttt{new[]}, \texttt{delete[]}, \texttt{()}, and \texttt{[]} are formed from more than one token. — end note]

Both the unary and binary forms of

\begin{verbatim}
+ - * &
\end{verbatim}

can be overloaded.

The following operators cannot be overloaded:

\begin{verbatim}
. .* :: ?:
\end{verbatim}
nor can the preprocessing symbols # and ## (Clause 19).

Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (16.5.1 – 16.5.7). They can be explicitly called, however, using the operator-function-id as the name of the function in the function call syntax (8.5.1.2). [Example:

```c
complex z = a.operator+(b);  // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

—end example]

5 The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete` and `operator delete[]`, are described completely in 6.6.4.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 6.6.4.4.

6 An operator function shall either be a non-static member function or be a non-member function that has at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators `=`, (unary) `&`, and `, (comma), predefined for each type, can be changed for specific class and enumeration types by defining operator functions that implement these operators. Operator functions are inherited in the same manner as other base class functions.

7 The identities among certain predefined operators applied to basic types (for example, `++a ≡ a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.

8 An operator function cannot have default arguments (11.3.6), 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.

9 Operators not mentioned explicitly in subclauses 16.5.3 through 16.5.7 act as ordinary unary and binary operators obeying the rules of 16.5.1 or 16.5.2.

### 16.5.1 Unary operators

A prefix unary operator shall be implemented by a non-static member function (12.2.1) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator `@`, `@x` can be interpreted as `x.operator@()` or `operator@(x)`. If both forms of the operator function have been declared, the rules in 16.3.1.2 determine which, if any, interpretation is used. See 16.5.7 for an explanation of the postfix unary operators `++` and `--`.

2 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]

### 16.5.2 Binary operators

A binary operator shall be implemented either by a non-static member function (12.2.1) with one parameter or by a non-member function with two parameters. Thus, for any binary operator `@`, `x@y` can be interpreted as `x.operator@(y)` or `operator@(x,y)`. If both forms of the operator function have been declared, the rules in 16.3.1.2 determine which, if any, interpretation is used.

### 16.5.3 Assignment

An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator `operator=` is implicitly declared for a class if not declared by the user (15.8), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

2 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:

```c
struct B {
    virtual int operator= (int);
    virtual B& operator= (const B&);
};
struct D : B {
    virtual int operator= (int);
```
16.5.4 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 \text{postfix-expression} evaluates to a class object and the possibly empty \text{expression-list} matches the
parameter list of an \text{operator()} member function of the class. Thus, a call \text{x(arg1,\ldots)} is interpreted as
\text{x.operator()}(arg1, \ldots) for a class object \text{x} of type \text{T} if \text{T::operator()}(T1, T2, T3) exists and if the
operator is selected as the best match function by the overload resolution mechanism (16.3.3).

16.5.5 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 \text{x[y]} is interpreted as \text{x.operator[]}(y) for a class object \text{x} of type \text{T} if
\text{T::operator[]}(T1) exists and if the operator is selected as the best match function by the overload resolution
mechanism (16.3.3). [Example:

\[
\text{struct X {}
    Z operator[](std::initializer_list<int>);}
\];
X x;
x[{1,2,3}] = 7; // OK: meaning x.operator[](1,2,3)
int a[10];
a[{1,2,3}] = 7; // error: built-in subscript operator
\]

16.5.6 Class member access

operator--> shall be a non-static member function taking no parameters. It implements the class member
access syntax that uses \text{->}.

\[
\text{postfix-expression} \rightarrow \text{template}_{\text{opt}} \ id-expression
\text{postfix-expression} \rightarrow \text{pseudo-destructor-name}
\]

An expression \text{x->m} is interpreted as \text{(x.operator-->())->m} for a class object \text{x} of type \text{T} if \text{T::operator-->()}
exists and if the operator is selected as the best match function by the overload resolution mechanism (16.3).

16.5.7 Increment and decrement

The user-defined function called \text{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 \text{int}) or a non-member function with two parameters (the second
of which shall be of type \text{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.

[Example:

```cpp
struct X {
  X& operator++(); // prefix ++a
  X operator++(int); // postfix a++
};

struct Y {
  Y& operator++(Y&); // prefix ++b
  Y operator++(Y&, int); // postfix b++
};

void f(X a, Y b) {
  ++a;           // a.operator++();
  a++;           // a.operator++(0);
  ++b;           // operator++(b);
  b++;           // operator++(b, 0);

  a.operator++(); // explicit call: like ++a;
  a.operator++(0); // explicit call: like a++;
  operator++(b);  // explicit call: like ++b;
  operator++(b, 0); // explicit call: like b++;
}
```
— end example]

The prefix and postfix decrement operators -- are handled analogously.

16.5.8 User-defined literals

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 (14.3)), an explicit instantiation or specialization of a function template, or a `using-declaration` (10.3.3). A function declared with a `literal-operator-id` is a literal operator. A function template declared with a `literal-operator-id` is a literal operator template.

The declaration of a literal operator shall have a `parameter-declaration-clause` equivalent to one of the following:

- `const char*`
- `unsigned long long int`
- `long double`
- `char`
- `wchar_t`
- `char16_t`
- `char32_t`
- `const char*, std::size_t`
- `const wchar_t*, std::size_t`
- `const char16_t*, std::size_t`
- `const char32_t*, std::size_t`

If a parameter has a default argument (11.3.6), the program is ill-formed.

A raw literal operator is a literal operator with a single parameter whose type is `const char*`.

135) Calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.
N4713

© ISO/IEC

5

The declaration of a literal operator template shall have an empty parameter-declaration-clause and its
template-parameter-list shall have a single template-parameter that is a non-type template parameter pack
(17.6.3) with element type char.

6

Literal operators and literal operator templates shall not have C language linkage.

7

8

[ 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 or external linkage, they can be called explicitly, their addresses can be taken, etc.
— end note ]
[ Example:
void operator "" _km(long double);
string operator "" _i18n(const char*, std::size_t);
template <char...> double operator "" _\u03C0();
float operator ""_e(const char*);
float operator ""E(const char*);
double operator""_Bq(long double);
double operator"" _Bq(long double);
float operator " " B(const char*);
string operator "" 5X(const char*, std::size_t);
double operator "" _miles(double);
template <char...> int operator "" _j(const char*);
extern "C" void operator "" _m(long double);

//
//
//
//
//
//
//
//
//
//
//
//

OK
OK
OK: UCN for lowercase pi
OK
error: reserved literal suffix (20.5.4.3.5, 5.13.8)
OK: does not use the reserved identifier _Bq (5.10)
uses the reserved identifier _Bq (5.10)
error: non-empty string-literal
error: invalid literal suffix identifier
error: invalid parameter-declaration-clause
error: invalid parameter-declaration-clause
error: C language linkage

— end example ]

16.6

Built-in operators

[over.built]

1

The candidate operator functions that represent the built-in operators defined in 8.5 are specified in this
subclause. These candidate functions participate in the operator overload resolution process as described
in 16.3.1.2 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 16.3.1.2, after a built-in operator is selected by overload resolution
the expression is subject to the requirements for the built-in operator given in 8.5, 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 ]

2

In this subclause, the term promoted integral type is used to refer to those integral types which are preserved
by integral promotion (7.6) (including e.g. int and long but excluding e.g. char). Similarly, the term
promoted arithmetic type refers to floating 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 ]

3

In the remainder of this subclause, vq represents either volatile or no cv-qualifier.

4

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);

5

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);

6

For every pair (T , vq), where T is a cv-qualified or cv-unqualified object type, there exist candidate operator
functions of the form

§ 16.6

303


For every cv-qualified or cv-unqualified object type \( T \), there exist candidate operator functions of the form

\[ T \& \text{operator}++(T\&); \]
\[ T \& \text{operator}--(T\&); \]
\[ T \& \text{operator}+(T\&, \text{int}); \]
\[ T \& \text{operator}-(T\&, \text{int}); \]

7 For every cv-qualified or cv-unqualified object type \( T \), there exist candidate operator functions of the form

\[ \text{int} \text{operator}++(T*); \]
\[ \text{int} \text{operator}--(T*); \]
\[ \text{int} \text{operator}+(T*, \text{int}); \]
\[ \text{int} \text{operator}-(T*, \text{int}); \]

8 For every function type \( T \) that does not have cv-qualifiers or a ref-qualifier, there exist candidate operator functions of the form

\[ \text{int} \text{operator}*(T*); \]
\[ \text{int} \text{operator}*(T*); \]

9 For every type \( T \) there exist candidate operator functions of the form

\[ \text{int} \text{operator}*(T*); \]
\[ \text{int} \text{operator}+(T*); \]
\[ \text{int} \text{operator}-(T*); \]

10 For every promoted arithmetic type \( T \), there exist candidate operator functions of the form

\[ \text{int} \text{operator}*(T*); \]
\[ \text{int} \text{operator}+(T*); \]
\[ \text{int} \text{operator}-(T*); \]

11 For every promoted integral type \( T \), there exist candidate operator functions of the form

\[ \text{int} \text{operator}*(T*); \]
\[ \text{int} \text{operator}+(T*); \]
\[ \text{int} \text{operator}-(T*); \]

12 For every quintuple \((C1, C2, T, cv1, cv2)\), where \( C2 \) is a class type, \( C1 \) is the same type as \( C2 \) or is a derived class of \( C2 \), and \( T \) is an object type or a function type, there exist candidate operator functions of the form

\[ \text{cv12} \text{operator}-->*(cv1 \& C1, cv2 \& C2::*); \]

where \( \text{cv12} \) is the union of \( \text{cv1} \) and \( \text{cv2} \). The return type is shown for exposition only; see 8.5.4 for the determination of the operator’s result type.

13 For every pair of promoted arithmetic types \( L \) and \( R \), there exist candidate operator functions of the form

\[ \text{LR} \text{operator}*(L, R); \]
\[ \text{LR} \text{operator}/(L, R); \]
\[ \text{LR} \text{operator}+(L, R); \]
\[ \text{LR} \text{operator}-(L, R); \]
\[ \text{bool} \text{operator}<(L, R); \]
\[ \text{bool} \text{operator}<=(L, R); \]
\[ \text{bool} \text{operator}>(L, R); \]
\[ \text{bool} \text{operator}>=((L, R); \]
\[ \text{bool} \text{operator}!(L, R); \]

where \( \text{LR} \) is the result of the usual arithmetic conversions (8.3) between types \( L \) and \( R \).

14 For every integral type \( T \) there exists a candidate operator function of the form

\[ \text{std::strong_ordering} \text{operator}=>(T, T); \]

15 For every pair of floating-point types \( L \) and \( R \), there exists a candidate operator function of the form

\[ \text{std::partial_ordering} \text{operator}=>(L, R); \]

16 For every cv-qualified or cv-unqualified object type \( T \) there exist candidate operator functions of the form

\[ \text{int} \text{operator}*(T*, \text{std::ptrdiff_t}); \]
\[ \text{int} \text{operator}[](T*, \text{std::ptrdiff_t}); \]
\[ \text{int} \text{operator}+(T*, \text{std::ptrdiff_t}); \]
\[ \text{int} \text{operator}-(T*, \text{std::ptrdiff_t}); \]

17 For every \( T \), where \( T \) is a pointer to object type, there exist candidate operator functions of the form

\[ \text{std::ptrdiff_t} \text{operator}-(T, T); \]

18 For every \( T \), where \( T \) is an enumeration type or a pointer type, there exist candidate operator functions of the form

\[ \text{bool} \text{operator}<(T, T); \]
\[ \text{bool} \text{operator}>(T, T); \]
\[ \text{bool} \text{operator}<=(T, T); \]
\[ \text{bool} \text{operator}>=(T, T); \]
\[ \text{bool} \text{operator}!(T, T); \]

§ 16.6 304
For every \( T \), where \( T \) is a pointer-to-member type or `std::nullptr_t`, there exist candidate operator functions of the form

\[
\begin{align*}
    \text{bool} & \quad \text{operator}!= (T, T); \\
    R & \quad \text{operator}<> (T, T);
\end{align*}
\]

where \( R \) is the result type specified in 8.5.8.

For every pair of promoted integral types \( L \) and \( R \), there exist candidate operator functions of the form

\[
\begin{align*}
    L & \quad \text{operator}%(L, R); \\
    LR & \quad \text{operator}k(L, R); \\
    LR & \quad \text{operator}^-(L, R); \\
    LR & \quad \text{operator}!(L, R); \\
    L & \quad \text{operator}<<(L, R); \\
    L & \quad \text{operator}>>(L, R);
\end{align*}
\]

where \( LR \) is the result of the usual arithmetic conversions (8.3) between types \( L \) and \( R \).

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

\[
\begin{align*}
    vq Lk & \quad \text{operator}=(vq Lk, R); \\
    vq Lk & \quad \text{operator}==(vq Lk, R); \\
    vq Lk & \quad \text{operator}=(vq Lk, R); \\
    vq Lk & \quad \text{operator}+=(vq Lk, R); \\
    vq Lk & \quad \text{operator}=(vq Lk, R);
\end{align*}
\]

For every pair \((T, vq)\), where \( T \) is any type, there exist candidate operator functions of the form

\[
T*vqk \quad \text{operator}=(T*vqk, T*);
\]

For every pair \((T, vq)\), where \( T \) is an enumeration or pointer-to-member type, there exist candidate operator functions of the form

\[
vq Tk \quad \text{operator}=(vq Tk, T);
\]

For every pair \((T, vq)\), where \( T \) is a cv-qualified or cv-unqualified object type, there exist candidate operator functions of the form

\[
T*vqk \quad \text{operator}=(T*vqk, std::ptrdiff_t); \\
T*vqk \quad \text{operator}==(T*vqk, std::ptrdiff_t);
\]

For every triple \((L, vq, R)\), where \( L \) is an integral type, and \( R \) is a promoted integral type, there exist candidate operator functions of the form

\[
\begin{align*}
    vq Lk & \quad \text{operator} %= (vq Lk, R); \\
    vq Lk & \quad \text{operator}<<=(vq Lk, R); \\
    vq Lk & \quad \text{operator}>>=(vq Lk, R); \\
    vq Lk & \quad \text{operator}^==(vq Lk, R); \\
    vq Lk & \quad \text{operator}^-= (vq Lk, R);
\end{align*}
\]

There also exist candidate operator functions of the form

\[
\begin{align*}
    \text{bool} & \quad \text{operator}!(bool); \\
    \text{bool} & \quad \text{operator}&& (bool, bool); \\
    \text{bool} & \quad \text{operator}||(bool, bool);
\end{align*}
\]

For every pair of promoted arithmetic types \( L \) and \( R \), there exist candidate operator functions of the form

\[
LR \quad \text{operator}?: (bool, L, R);
\]

where \( LR \) is the result of the usual arithmetic conversions (8.3) between types \( L \) and \( R \). [Note: As with all these descriptions of candidate functions, this declaration serves only to describe the built-in operator for purposes of overload resolution. The operator “?::” cannot be overloaded. — end note]

For every type \( T \), where \( T \) is a pointer, pointer-to-member, or scoped enumeration type, there exist candidate operator functions of the form

\[
T \quad \text{operator}?: (bool, T, T);
\]
17 Templates

A template defines a family of classes, functions, or variables, an alias for a family of types, or a concept.

template-declaration:
  template-head declaration
  template-head concept-definition

template-head:
  template < template-parameter-list > requires-clause_opt

template-parameter-list:
  template-parameter
  template-parameter-list , template-parameter

requires-clause:
  requires constraint-logical-or-expression

constraint-logical-or-expression:
  constraint-logical-and-expression
  constraint-logical-or-expression || constraint-logical-and-expression

constraint-logical-and-expression:
  primary-expression
  constraint-logical-and-expression && primary-expression

concept-definition:
  concept concept-name = constraint-expression;

concept-name:
  identifier

[Note: The > token following the template-parameter-list of a template-declaration may be the product of replacing a >> token by two consecutive > tokens (17.2). — 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:

```cpp
template<class T>
  constexpr T pi = T(3.1415926535897932385L);
template<class T>
  T circular_area(T r) {
    return pi<T> * r * r;
  }

struct matrix_constants {
  template<class T>
    using pauli = hermitian_matrix<T, 2>;
  template<class T>
    constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
  template<class T>
    constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
```
]
template<class T>
constexpr pauli<T> sigma3 = {{ 1, 0 }, { 0, -1 }};

— end example

4 A template-declaration can appear only as a namespace scope or class scope 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 (17.6.5). — end note]

5 In a template-declaration, explicit specialization, or explicit instantiation the init-declarator-list in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.

6 A template name has linkage (6.5). 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 (17.8.3), 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 (17.6) and must also obey the one-definition rule. — end note

7 A class template shall not have the same name as any other template, class, function, variable, enumeration, enumerator, namespace, or type in the same scope (6.3), except as specified in 17.6.5. Except that a function template can be overloaded either by non-template functions (11.3.5) with the same name or by other function templates with the same name (17.9.3), a template name declared in namespace scope or in class scope shall be unique in that scope.

8 A templated entity is

(8.1) — a template,
(8.2) — an entity defined (6.1) or created (15.2) in a templated entity,
(8.3) — a member of a templated entity,
(8.4) — an enumerator for an enumeration that is a templated entity, or
(8.5) — the closure type of a lambda-expression (8.4.5.1) appearing in the declaration of a templated entity.

[ Note: A local class, a local variable, or a friend function defined in a templated entity is a templated entity. — end note]

9 A template-declaration is written in terms of its template parameters. The optional requires-clause following a template-parameter-list allows the specification of constraints (17.4.2) on template arguments (17.3). 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 8). [Note: The expression in a requires-clause uses a restricted grammar to avoid ambiguities. Parentheses can be used to specify arbitrary expressions in a requires-clause. [Example:

```cpp
template<int N> requires N == sizeof new unsigned short
int f();   // error: parentheses required around == expression

— end example]
```

— end note

10 A 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 (17.8.1) unless the corresponding specialization is explicitly instantiated (17.8.2) in some translation unit; no diagnostic is required.

17.1 Template parameters

The syntax for template-parameters is:

```
template-parameter:
  type-parameter
  parameter-declaration
  constrained-parameter
```

§ 17.1
type-parameter:
  type-parameter-key ...opt identifieropt
  type-parameter-key identifieropt = type-id
  template-head type-parameter-key ...opt identifieropt
  template-head type-parameter-key identifieropt = id-expression

type-parameter-key:
  class
typename

constrained-parameter:
  qualified-concept-name ... identifieropt
  qualified-concept-name identifieropt default-template-argumentopt

qualified-concept-name:
  nested-name-specifieropt concept-name
  nested-name-specifieropt partial-concept-id

partial-concept-id:
  concept-name < template-argument-listopt >

default-template-argument:
  = type-id
  = id-expression
  = initializer-clause

[Note: The \textgreater \textgreater token following the template-parameter-list of a type-parameter may be the product of replacing a \textgreater \textgreater \textgreater token by two consecutive \textgreater tokens (17.2).] — end note]

2 There is no semantic difference between \texttt{class} and \texttt{typename} in a type-parameter-key. \texttt{typename} followed by an unqualified-\texttt{id} names a template type parameter. \texttt{typename} followed by a qualified-\texttt{id} denotes the type in a non-type\textsuperscript{136} parameter-declaration. A template-parameter of the form \texttt{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 \texttt{f} has a type-parameter called \texttt{T}, rather than an unnamed non-type template-parameter of class \texttt{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 \texttt{typedef-name} (if declared without \texttt{template}) or \texttt{template-name} (if declared with \texttt{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) — integral or enumeration type,

(4.2) — pointer to object or pointer to function,

(4.3) — lvalue reference to object or lvalue reference to function,

(4.4) — pointer to member,

\textsuperscript{136} 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.

\section{17.1}
— std::nullptr_t, or
— a type that contains a placeholder type (10.1.7.4).

[Note: Other types are disallowed either explicitly below or implicitly by the rules governing the form of template-arguments (17.3). — end note] The top-level cv-qualifiers on the template-parameter are ignored when determining its type.

A non-type non-reference template-parameter is a prvalue. It shall not be assigned to or in any other way have its value changed. A non-type non-reference template-parameter cannot have its address taken. When a non-type non-reference template-parameter is used as an initializer for a reference, a temporary is always used. [Example:

```cpp
template<const X& x, int i> void f() {
    i++;
    // error: change of template-parameter value
    &x;
    // OK
    &i;
    // error: address of non-reference template-parameter
    int& ri = i;
    // error: non-const reference bound to temporary
    const int& rci = i;
    // OK: const reference bound to temporary
}
```

— end example]

A non-type template-parameter shall not be declared to have floating-point, class, or void type. [Example:

```cpp
template<double d> class X;  // error
template<double* pd> class Y; // OK
template<double& rd> class Z; // OK
```

— end example]

A non-type template-parameter of type “array of T” or of function type T is adjusted to be of type “pointer to T”. [Example:

```cpp
template<int* a> struct R { /* ... */ ;
template<int b[5]> struct S { /* ... */ ;
int p;
R<&p> v;  // OK
S<&p> x;  // OK due to parameter adjustment
int v[5];
R<<> y;  // OK due to implicit argument conversion
S<<> z;  // OK due to both adjustment and conversion
```

— end example]

A partial-concept-id is a concept-name followed by a sequence of template-arguments. These template arguments are used to form a constraint-expression as described below.

A constrained-parameter declares a template parameter whose kind (type, non-type, template) and type match that of the prototype parameter (17.6.8) of the concept designated by the qualified-concept-name in the constrained-parameter. Let X be the prototype parameter of the designated concept. The declared template parameter is determined by the kind of X (type, non-type, template) and the optional ellipsis in the constrained-parameter as follows.

(10.1) — If X is a type template-parameter, the declared parameter is a type template-parameter.
(10.2) — If X is a non-type template-parameter, the declared parameter is a non-type template-parameter having the same type as X.
(10.3) — If X is a template template-parameter, the declared parameter is a template template-parameter having the same template-parameter-list as X, excluding default template arguments.
(10.4) — If the qualified-concept-name is followed by an ellipsis, then the declared parameter is a template parameter pack (17.6.3).

[Example:

```cpp
template< typename T> concept C1 = true;
template<template< typename> class X> concept C2 = true;
template< typename N> concept C3 = true;
template< typename... Ts> concept C4 = true;
```
A constrained-parameter introduces a constraint-expression (17.4.2). The expression is derived from the qualified-concept-name \( Q \) in the constrained-parameter, its designated concept \( C \), and the declared template parameter \( P \).

11

A default template-argument is a template-argument (17.3) 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 (17.6.3). 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.

14

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 (11.3.6). [Example: }

\[
\begin{align*}
\text{template<} & \text{char... Cs> concept } C5 = \text{true;} \\
\text{template<C1 } & \text{T> void } f1(); \quad \text{OK;} \quad T \text{ is a type template-parameter} \\
\text{template<C2 } & \text{X> void } f2(); \quad \text{OK;} \quad X \text{ is a template with one type-parameter} \\
\text{template<C3 } & \text{N> void } f3(); \quad \text{OK;} \quad N \text{ has type int} \\
\text{template<C4... } & \text{Ts> void } f4(); \quad \text{OK;} \quad Tn \text{ is a template parameter pack of types} \\
\text{template<C4 } & \text{T> void } f5(); \quad \text{OK;} \quad T \text{ is a type template-parameter} \\
\text{template<C5... } & \text{Cs> void } f6(); \quad \text{OK;} \quad Cn \text{ is a template parameter pack of chars} \\
\end{align*}
\]
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 (11.3.5) of the function template or has a default argument (17.9.2). A template parameter of a deduction guide template (17.10) that does not have a default argument shall be deducible from the parameter-type-list of the deduction guide template. [Example:

```cpp
template<class T1 = int, class T2> class B; // error
```

// U can be neither deduced from the parameter-type-list nor specified
```cpp
template<class... T, class... U> void f() {} // error
template<class... T, class U> void g() {} // error
```

—end example]

A template-parameter shall not be given default arguments by two different declarations in the same scope. [Example:

```cpp
template<class T = int> class X;
template<class T = int> class X { /* ... */ }; // error
```

—end example]

When parsing a default template-argument for a non-type template-parameter, the first non-nested > is taken as the end of the template-parameter-list rather than a greater-than operator. [Example:

```cpp
template<int i = 3 > 4 >
class X { /* ... */ }; // syntax error
template<int i = (3 > 4) >
class Y { /* ... */ }; // OK
```

—end example]

A template-parameter of a template template-parameter is permitted to have a default template-argument. When such default arguments are specified, they apply to the template template-parameter in the scope of the template template-parameter. [Example:

```cpp
template <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]

If a template-parameter is a type-parameter with an ellipsis prior to its optional identifier or is a parameter-declaration that declares a parameter pack (11.3.5), then the template-parameter is a template parameter pack (17.6.3). A template parameter pack that is a parameter-declaration whose type contains one or more unexpanded parameter 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 parameter packs is a pack expansion. A template parameter pack that is a pack expansion shall not expand a 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>
// error: Values expands template type parameter
struct static_array;
// pack T within the same template parameter list

—end example

17.2 Names of template specializations

A template specialization (17.8) 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 finds either 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 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

template<class T> class Y { /* ... */};
Y<X<1>> x3; // OK, same as Y<X<1> > x3;
Y<X<6>1>> x4; // syntax error
Y<X<(6>>1)>> x5; // OK

—end example
```

The keyword template is said to appear at the top level in a qualified-id if it appears outside of a template-argument-list or decltype-specifier. In a qualified-id of a declarator-id or in a qualified-id formed by a class-head-name (Clause 12) or enum-head-name (10.2), the keyword template shall not appear at the top

---

137) A > that encloses the type-id of a dynamic_cast, static_cast, reinterpret_cast or const_cast, or which encloses the template-arguments of a subsequent template-id, is considered nested for the purpose of this description.
level. In a qualified-id used as the name in a typename-specifier (17.7), elaborated-type-specifier (10.1.7.3), using-declaration (10.3.3), or class-or-decltype (Clause 13), an optional keyword template appearing at the top level is ignored. In these contexts, a < token is always assumed to introduce a template-argument-list. In all other contexts, when naming a template specialization of a member of an unknown specialization (17.7.2.1), the member template name shall be prefixed by the keyword template. [Example:

```cpp
struct X {
    template< std::size_t > X* alloc();
    template< std::size_t > static X* adjust();
};

template<class T> void f(T* p) {
    T* p1 = p->alloc<200>(); // 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
}
```

—end example]

5 A name prefixed by the keyword template shall be a template-id or the name shall refer to a class template or an alias template. [Note: The keyword template may not be applied to non-template members of class templates. —end note] [Note: As is the case with the typename prefix, the template prefix is allowed in cases where it is not strictly necessary; i.e., when the nested-name-specifier or the expression on the left of the -> or . is not dependent on a template-parameter, or the use does not appear in the scope of a template. —end note] [Example:

```cpp
template <class T> struct A {
    void f(int);
    template <class U> void f(U);
};

// OK: T::template C names a class template:
template <class T, template <class X> class TT = T::template C> struct D { T::template C db; // illegal: do not construct template DDB
```

—end example]

6 A simple-template-id that names a class template specialization is a class-name (Clause 12).

7 A template-id that names an alias template specialization is a type-name.

8 When the template-name of a simple-template-id names a constrained non-function template or a constrained template-parameter, but not a member template that is a member of an unknown specialization (17.7), and all template-arguments in the simple-template-id are non-dependent (17.7.2.4), the associated constraints (17.4.2) of the constrained template shall be satisfied (17.4.1). [Example:

```cpp
template<typename T> concept C1 = sizeof(T) != sizeof(int);

// OK: T::template C names a class template:
template <class T, template <class X> class TT = T::template C> struct D { T::template C db; // illegal: do not construct template DDB
```

§ 17.2
17.3 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 (17.6.3), it will correspond to zero or more `template-argument`s.

```cpp
// Example:

template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};

Array<int> v1(20);
typedef std::complex<double> dcomplex; // std::complex is a standard library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
  v1[3] = 7;
  v2[3] = v3.elem(4) = dcomplex(7,8);
}
```

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`.

```cpp
// Example:

template<class T> void f();
template<int I> void f();

void g() {
  f<int>(); // int() is a type-id: call the first f()
}
```

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:

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`. ]
template<class T> class X {
  static T t;
};

class Y {
private:
  struct S { /* ... */ }; // OK: S is accessible
  X<S> x; // OK: 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:

```
template <template <class TT> class T> class A {
  typename T<int>::S s;
};

template <class U> class B {
private:
  struct S { /* ... */ }; // OK: S is accessible
};

A<B> 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:

```
template<class T = char> class String;
String** p; // OK: String<char>
String* q; // syntax error
```

```
template<class ... Elements> class Tuple;
Tuple** t; // OK: Elements is empty
Tuple* u; // syntax error
```

—end example]

5 An explicit destructor call (15.4) for an object that has a type that is a class template specialization may explicitly specify the template-arguments. [Example:

```
template<class T> struct A {
  A();
};

void f(A<int>** p, A<int>** q) {
  p->A<int>::~A(); // OK: destructor call
  q->A<int>::~A<int>(); // OK: destructor call
}
```

—end example]

6 If the use of a template-argument gives rise to an ill-formed construct in the instantiation of a template specialization, the program is ill-formed.

7 When name lookup for the name in a template-id finds an overload set, both non-template functions in the overload set and function templates in the overload set for which the template-arguments do not match the template-parameters are ignored. If none of the function templates have matching template-parameters, the program is ill-formed.

8 When a simple-template-id does not name a function, a default template-argument is implicitly instantiated (17.8.1) when the value of that default argument is needed. [Example:

```
template<typename T, typename U = int> struct S { }; // the type of p is S<bool, int>**
S<bool>** p;
```

The default argument for U is instantiated to form the type S<bool, int>**. — end example]
A template-argument followed by an ellipsis is a pack expansion (17.6.3).

17.3.1 Template type arguments

A template-argument for a template-parameter which is a type shall be a type-id.

Example:

```c
template <class T> class X { };
template <class T> void f(T t) { }
struct { } unnamed_obj;

void f() {
    struct A { };
    enum { e1 };
    typedef struct { } B;
    B b;
    X<A> x1; // OK
    X<A*> x2; // OK
    X<B> x3; // OK
    f(e1); // OK
    f(unnamed_obj); // OK
    f(b); // OK
}
```

—end example] [ Note: A template type argument may be an incomplete type (6.7). —end note]

17.3.2 Template non-type arguments

If the type of a template-parameter contains a placeholder type (10.1.7.4, 17.1), the deduced parameter type is determined from the type of the template-argument by placeholder type deduction (10.1.7.4.1). If a deduced parameter type is not permitted for a template-parameter declaration (17.1), the program is ill-formed.

A template-argument for a non-type template-parameter shall be a converted constant expression (8.6) of the type of the template-parameter. For a non-type template-parameter of reference or pointer type, the value of the constant expression shall not refer to (or for a pointer type, shall not be the address of):

1. a subobject (6.6.2),
2. a temporary object (15.2),
3. a string literal (5.13.5),
4. the result of a typeid expression (8.5.1.8), or
5. a predefined __func__ variable (11.4.1).

[ 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 (16.4). —end note]

Example:

```c
template<const int* pci> struct X { /* ... */ };
int ai[10];
X<ai> x1; // 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]  

[ Note: A string literal (5.13.5) is not an acceptable template-argument. [ Example:
  template<class T, const char* p> class X {
      /* ... */
  };  
  X<int, "Studebaker"> x1;  // error: string literal as template-argument  
  const char p[] = "Vivisectionist";
  X<int,p> x2;  // OK  
—end example ] — end note]

[ Note: The address of an array element or non-static data member is not an acceptable template-argument. [ Example:
  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<&a.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:
  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]

17.3.3 Template template arguments [temp.arg.template]  

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 (17.6.5) 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:
  template<class T> class A {     // primary template
      int x;
  };  
  template<class T> class A<T*> {     // partial specialization
      long x;
  };  
  template<template<class U> class V> class C {  
      V<int> y;
      V<int*> z;
  };
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 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 (17.6.6.1), and for template template-parameters, each of their corresponding template-parameters matches, recursively. When P’s template-head contains a template parameter pack (17.6.3), 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:

```cpp
template<template<class> class P> struct S { };  
template<C> struct X { };  
template<D> struct Y { };  
template<typename T> struct Z { };  
S<X> s1;  // OK, X and P have equivalent constraints
S<Y> s2;  // error: P is not at least as specialized as Y
S<Z> s3;  // OK, P is at least as specialized as Z
```

—end example
A template template-parameter $P$ is at least as specialized as a template template-argument $A$ if, given the following rewrite to two function templates, the function template corresponding to $P$ is at least as specialized as the function template corresponding to $A$ according to the partial ordering rules for function templates (17.6.6.2). Given an invented class template $X$ with the template-head of $A$ (including default arguments and requires-clause, if any):

1. Each of the two function templates has the same template parameters and requires-clause (if any), respectively, as $P$ or $A$.
2. Each function template has a single function parameter whose type is a specialization of $X$ with template arguments corresponding to the template parameters from the respective function template where, for each template parameter $PP$ in the template-head of the function template, a corresponding template argument $AA$ is formed. If $PP$ declares a parameter pack, then $AA$ is the pack expansion $PP$... (17.6.3); otherwise, $AA$ is the id-expression $PP$.

If the rewrite produces an invalid type, then $P$ is not at least as specialized as $A$.

17.4 Template constraints [temp.constr]

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 (17.4.1.1),
1.2. disjunctions (17.4.1.1), and
1.3. atomic constraints (17.4.1.2)

2. In order for a constrained template to be instantiated (17.8), its associated constraints (17.4.2) shall be satisfied as described in the following subsections. [Note: Forming the name of a specialization of a class template, a variable template, or an alias template (17.2) requires the satisfaction of its constraints. Overload resolution (16.3.2) requires the satisfaction of constraints on functions and function templates. — end note]

17.4.1 Logical operations [temp.constr.op]

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 $\wedge$ and disjunction is spelled using the symbol $\vee$. The operands of these operations are called the left and right operands. In the constraint $A \wedge B$, $A$ is the left operand, and $B$ is the right operand. — end note]

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 satisfied, the disjunction is satisfied. Otherwise, the disjunction is satisfied if and only if the second operand is satisfied.

4. [Example:

```cpp
template<typename T>
constexpr bool get_value() { return T::value; }

template<typename T>
requires (sizeof(T) > 1) && get_value<T>()
void f(T); // has associated constraint sizeof(T) > 1 \&\& get_value<T>()

void f(int);

f('a'); // OK: calls f(int)
```

In the satisfaction of the associated constraints (17.4.2) of $f$, the constraint $\text{sizeof(char)} > 1$ is not satisfied; the second operand is not checked for satisfaction. — end example]
17.4.1.2 Atomic constraints

An atomic constraint is formed from an expression \( E \) and a mapping from the template parameters that appear within \( E \) to template arguments involving the template parameters of the constrained entity, called the parameter mapping (17.4.2). [Note: Atomic constraints are formed by constraint normalization (17.4.3). \( E \) is never a logical AND expression (8.5.14) nor a logical OR expression (8.5.15). — end note]

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 17.6.6.1.

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 (7.1) is performed if necessary, and \( E \) shall be a constant expression of type \( \text{bool} \). The constraint is satisfied if and only if evaluation of \( E \) results in \( \text{true} \).

Example:

```cpp
template<typename T> concept C =
    sizeof(T) == 4 && !true;  // requires atomic constraints sizeof(T) == 4 and !true

template<typename T> struct S {
    constexpr operator bool() const { return true; }
};

template<typename T> requires (S<T>{})
    void f(T);  // #1
    void f(int);  // #2

void g() {
    f(0);       // error: expression S<int>{} does not have type bool
    // while checking satisfaction of deduced arguments of #1;
    // call is ill-formed even though #2 is a better match
}
```

17.4.2 Constrained declarations

A template declaration (Clause 17) or function declaration (11.3.5) can be constrained by the use of a requires-clause. This allows the specification of constraints for that declaration as an expression:

```
constraint-expression:
    logical-or-expression
```

2 Constraints can also be associated with a declaration through the use of constrained-parameters in a template-parameter-list. Each of these forms introduces additional constraint-expressions that are used to constrain the declaration.

3 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 (17.4.3) of that expression.
3. Otherwise, the associated constraints are the normal form of a logical AND expression (8.5.14) whose operands are in the following order:
   1. the constraint-expression introduced by each constrained-parameter (17.1) 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 17), and
   3. the constraint-expression introduced by a trailing requires-clause (Clause 11) of a function declaration (11.3.5).

The formation of the associated constraints establishes the order in which constraints are instantiated when checking for satisfaction (17.4.1). [Example:

```cpp
template<typename T> concept C = true;
```
The functions `f1`, `f2`, and `f3` have the associated constraint `C<T>`.

The associated constraints of `f4` and `f5` are `C1<T> ∧ C2<T>`. The associated constraints of `f6` and `f7` are `C1<T> ∧ C2<T>`, and those of `f7` are `C2<T> ∧ C1<T>`. — end example]

17.4.3 Constraint normalization [temp.constr.normal]

The normal form of an expression `E` is a constraint (17.4.1) 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 `E1 || E2` is the disjunction (17.4.1.1) of the normal forms of `E1` and `E2`.
3. The normal form of an expression `E1 && E2` is the conjunction of the normal forms of `E1` and `E2`.
4. The normal form of an `id-expression` of the form `C<A1, A2, ..., An>`, where `C` names a concept, is the normal form of the `constraint-expression` of `C`, after substituting `A1, A2, ..., An` for `C`'s respective template parameters in the parameter mappings in each atomic constraint. If any such substitution results in an invalid type or expression, the program is ill-formed; no diagnostic is required.

[Example:

```cpp
template<typename T> concept A = T::value || true;
template<typename U> concept B = A<U*>;
template<typename V> concept C = B<V&>;
```

Normalization of B's `constraint-expression` is valid and results in `T::value` (with the mapping `T`↦→`U*`) ∨ true (with an empty mapping), despite the expression `T::value` being ill-formed for a pointer type `T`. Normalization of C's `constraint-expression` results in the program being ill-formed, because it would form the invalid type `T&*` in the parameter mapping. — end example]

5. The normal form of any other expression `E` is the atomic constraint whose expression is `E` and whose parameter mapping is the identity mapping.

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 (17.4.1) of a declaration and when evaluating the value of an `id-expression` that names a concept specialization (8.4.4). — end note]

[Example:

```cpp
template<typename T> concept C1 = sizeof(T) == 1;
template<typename T> concept C2 = C1<T>() && 1 == 2;
template<typename T> concept C3 = requires { typename T::type; };
template<typename T> concept C4 = requires (T x) { ++x; }
```

The associated constraints of #1 are `sizeof(T) == 1` (with mapping `T`↦→`U`) ∧ 1 == 2.
The associated constraints of #2 are `requires { typename T::type; }` (with mapping `T`↦→`U`).
The associated constraints of #3 are `requires (T x) { ++x; }` (with mapping `T`↦→`U`). — end example]
17.4.4 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\(^{139} \) of \( P \), \( P_i \) subsumes every conjunctive clause \( Q_j \) in the conjunctive normal form\(^{140} \) of \( Q \), where

1. A disjunctive clause \( P_i \) subsumes a conjunctive clause \( Q_j \) if and only if there exists an atomic constraint \( P_i a \) in \( P_i \) such that \( P_i a \) subsumes every conjunctive clause \( Q_j b \) in \( Q_j \);

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 17.4.1.2.

\[ \text{Example:} \text{ Let } A \text{ and } B \text{ be atomic constraints (17.4.1.2). The constraint } A \land B \text{ subsumes } A \text{, but } A \text{ does not subsume } A \land B. \text{ The constraint } A \text{ subsumes } A \lor B, \text{ but } A \lor B \text{ does not subsume } A. \text{ Also note that every constraint subsumes itself. } \text{— end example} \]

2. The subsumption relation defines a partial ordering on constraints. This partial ordering is used to determine

1. the best viable candidate of non-template functions (16.3.3),
2. the address of a non-template function (16.4),
3. the matching of template template arguments (17.3.3),
4. the partial ordering of class template specializations (17.6.5.2), and
5. the partial ordering of function templates (17.6.6.2).

— end note

A declaration \( D_1 \) is at least as constrained as a declaration \( D_2 \) if

1. \( D_1 \) and \( D_2 \) are both constrained declarations and \( D_1 \)'s associated constraints subsume those of \( D_2 \); or
2. \( D_2 \) has no associated constraints.

A declaration \( D_1 \) is more constrained than another declaration \( D_2 \) when \( D_1 \) is at least as constrained as \( D_2 \), and \( D_2 \) is not at least as constrained as \( D_1 \). [Example:

\[
\begin{align*}
\text{template<typename T> concept C1 = requires(T t) { --t; };} \\
\text{template<typename T> concept C2 = C1<T> && requires(T t) { *t; };} \\
\text{template<C1 T> void f(T);} & \text{ (#1)} \\
\text{template<C2 T> void f(T);} & \text{ (#2)} \\
\text{template<typename T> void g(T);} & \text{ (#3)} \\
\text{template<C1 T> void g(T);} & \text{ (#4)} \\
\text{f(0);} & \text{ (#4 selects #1)} \\
\text{f((int*)0);} & \text{ (#2 selects #2)} \\
\text{g(true);} & \text{ (#3 selects #3 because C1<bool> is not satisfied} \\
\text{g(0);} & \text{ (#4 selects #4)} \\
\end{align*}
\]

— end example]

17.5 Type equivalence

Two template-ids refer to the same class, function, or variable if

1. Their template-names, operator-function-ids, or literal-operator-ids refer to the same template and
2. Their corresponding type template-arguments are the same type and
3. Their corresponding non-type template arguments of integral or enumeration type have identical values and
4. Their corresponding non-type template-arguments of pointer type refer to the same object or function or are both the null pointer value and

\(^{139}\) 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]

\(^{140}\) 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 \( A \lor (B \land C) \). — end example]
— 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

— their corresponding non-type template-arguments of reference type refer to the same object or function and

— their corresponding template template-arguments refer to the same template.

Example:

```c++
template<class E, int size> class buffer { /* ... */ };  
buffer<char,2*512> x;  
buffer<char,1024> y;  
```
decrees x and y to be of the same type, and

```c++
template<class T, void(*err_fct)()> class list { /* ... */ };  
list<int,&error_handler1> x1;  
list<int,&error_handler2> x2;  
list<int,&error_handler2> x3;  
list<char,&error_handler2> x4;  
```
decrees x2 and x3 to be of the same type. Their type differs from the types of x1 and x4.

```c++
template<class T> struct X { };  
template<class> struct Y { };  
template<class T> using Z = Y<T>;  
X<Y<int> > y;  
X<Z<int> > z;  
```
decrees y and z to be of the same type. — end example]

2 If an expression e is type-dependent (17.7.2.2), decltype(e) denotes a unique dependent type. Two such decltype-specifiers refer to the same type only if their expressions are equivalent (17.6.6.1). [ Note: However, such a type may be aliased, e.g., by a typedef-name. — end note ]

17.6 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:

```c++
template<class T1, class T2, int I> class A<T1, T2, I> { }; // error  
template<class T1, int I> void sort<T1, I>(T1 data[I]); // error  
```
— end example] [ Note: However, this syntax is allowed in class template partial specializations (17.6.5). — end note ]

2 For purposes of name lookup and instantiation, default arguments, partial-concept-ids, requires-clauses (Clause 17), and noexcept-specifiers of function templates and of member functions of class templates are considered definitions; each default argument, partial-concept-ids, requires-clause, or noexcept-specifier is a separate definition which is unrelated to the templated function definition or to any other default arguments partial-concept-ids, requires-clauses, or noexcept-specifiers. For the purpose of instantiation, the substatements of a constexpr if statement (9.4.1) 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.

17.6.1 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:

```c++
template<class T> class Array {  
    T* v;  
    int sz;  
public:  
    explicit Array(int);  
    T& operator[](int);  
    T& elem(int i) { return v[i]; }  
};  
```

§ 17.6.1 323
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 (17.6.6.1). 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 f1();
    void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { } // OK
template<class T2, class T1> void A<T1,T2>::f2() { } // error

template<class ... Types> struct B {
    void f3();
    void f4();
};

template<class ... Types> void B<Types ...>::f3() { } // OK
template<class ... Types> void B<Types>::f4() { } // error

template<typename T> concept C = true;
template<typename T> concept D = true;

template<C T> struct S {
    void f();
    void g();
    void h();
    template<typename U> struct Inner;
    template<typename T> void S<T>::f() { }
    // OK: template-heads match
    template<typename T> void S<T>::g() { }
    // error: no matching declaration for S<T>
    template<typename T> requires C<T>
    // error (no diagnostic required): template-heads are
    // functionally equivalent but not equivalent

    template<C X> template<typename Y> struct S<X>::Inner { }; // OK

    // end example]
```

In a redeclaration, partial specialization, explicit specialization or explicit instantiation of a class template, the class-key shall agree in kind with the original class template declaration (10.1.7.3).

**17.6.1.1 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:

```cpp
template<class T> class Array {
    T* v;
    int sz;
public:
    explicit Array(int);
    T& operator[](int);  // T& elem(int i) { return v[i]; } }
};
```

declares three function templates. 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]

The template-arguments for a member function of a class template are determined by the template-arguments of the type of the object for which the member function is called. [Example: The template-argument for Array<T>::operator[]() will be determined by the Array to which the subscripting operation is applied.]

Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7; // Array<int>::operator[]()
v2[3] = dcomplex(7,8); // Array<dcomplex>::operator[]()
—end example]

17.6.1.2 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 (17.8.1). For example,]

template<class T> struct A {
    class B;
};
A<int>::B* b1; // OK: requires A to be defined but not A::B
template<class T> class A<T>::B { }; // OK: requires A::B to be defined
—end note]

17.6.1.3 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:]

template<class T> class X {
    static T s;
};
template<class T> T X<T>::s = 0;

struct limits {
    template<class T>
        static const T min; // declaration
};

template<class T>
    const T limits::min = { }; // definition
—end example]
An explicit specialization of a static data member declared as an array of unknown bound can have a different bound from its definition, if any. [Example:

```cpp
template <class T> struct A {
    static int i[];
};
template <class T> int A<T>::i[] = { 1 }; // OK: 1 element
```
—end example]

### 17.6.1.4 Enumeration members of class templates

An enumeration member of a class template may be defined outside the class template definition. [Example:

```cpp
template<class T> struct A {
    enum E : T;
};
A<int> a;
template<class T> enum A<T>::E : T { e1, e2 };
A<int>::E e = A<int>::e1;
```
—end example]

### 17.6.2 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 (17.6.6.1). [Example:

```cpp
template<class T> struct string {
    template<class T2> int compare(const T2&);
    template<class T2> string(const string<T2>& s) { /* ... */ }
};
```
—end example]

```cpp
template<typename T> concept C1 = true;
template<typename T> concept C2 = sizeof(T) <= 4;
template<C1 T> struct S {
    template<C2 U> void f(U);
    template<C2 U> void g(U);
};
```

```cpp
template<C1 T> template<C2 U>
void S<T>::f(U) { } // OK
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 (Clause 14) apply to member template names. A destructor shall not be a member template. A non-template member function (11.3.5) with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can both be declared in a class. When both exist, a use of that name and type refers to the non-template member unless an explicit template argument list is supplied. [Example:

```cpp
template <class T> struct A {
    void f(int);
    template <class T2> void f(T2);
};
```
template <> void A<int>::f(int) { }  // non-template member function

template <> template <> void A<int>::f<> (int) { }  // member function template specialization

int main() {
    A<char> ac;
    ac.f(1);           // non-template
    ac.f('c');        // template
    ac.f<> (1);        // template
}

— end example

A member function template shall not be virtual.  [Example:

```cpp
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:

```cpp
class B {
    virtual void f(int);
};

class D : public B {
    template <class T> void f(T); // does not override B::f(int)
    void f(int i) { f<> (i); }    // overriding function that calls the template instantiation
};
```

— end example

A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type.  [Example:

```cpp
struct A {
    template <class T> operator T*();
};

template <class T> A::operator T*() { return 0; }
template <> A::operator char*() { return 0; }    // specialization
A::operator void*();                           // explicit instantiation
```

```cpp
int main() {
    A a;
    int* ip;
    ip = a.operator int*();                   // explicit call to template operator A::operator int*()
}
```

— end example]  [Note: Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. — 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 (17.9.2.3), 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 (16.3.3.2) and partial ordering (17.6.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

17.6.3 Variadic templates  [temp.variadic]

A template parameter pack is a template parameter that accepts zero or more template arguments.  [Example:

```cpp
template<class ... Types> struct Tuple { }
```

§ 17.6.3
Tuple<> t0; // Types contains no arguments
Tuple<int> t1; // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float
Tuple<> error; // error: 0 is not a type

— end example]

2 A **function parameter pack** is a function parameter that accepts zero or more function arguments. [Example:

```cpp
template<class ... Types> void f(Types ... args);
```

```cpp
f(); // OK: args contains no arguments
f(1); // OK: args contains one argument: int
f(2, 1.0); // OK: args contains two arguments: int and double

— end example]

3 A **parameter pack** is either a template parameter pack or a function parameter pack.

4 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:

(4.1) — In a function parameter pack (11.3.5); the pattern is the **parameter-declaration** without the ellipsis.

(4.2) — In a **using-declaration** (10.3.3); the pattern is a **using-declarator**.

(4.3) — In a template parameter pack that is a pack expansion (17.1):
   - 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** with a **template-parameter-list**; the pattern is the corresponding **type-parameter** without the ellipsis.

(4.4) — In an **initializer-list** (11.6); the pattern is an **initializer-clause**.

(4.5) — In a **base-specifier-list** (Clause 13); the pattern is a **base-specifier**.

(4.6) — In a **mem-initializer-list** (15.6.2) for a **mem-initializer** whose **mem-initializer-id** denotes a base class; the pattern is the **mem-initializer**.

(4.7) — In a **template-argument-list** (17.3); the pattern is a **template-argument**.

(4.8) — In an **attribute-list** (10.6.1); the pattern is an **attribute**.

(4.9) — In an **alignment-specifier** (10.6.2); the pattern is the **alignment-specifier** without the ellipsis.

(4.10) — In a **capture-list** (8.4.5); the pattern is a **capture**.

(4.11) — In a **sizeof...** expression (8.5.2.3); the pattern is an **identifier**.

(4.12) — In a **fold-expression** (8.4.6); the pattern is the **cast-expression** that contains an unexpanded parameter pack.

[Example:

```cpp
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
    f(&rest ...); // “&rest ...” is a pack expansion; “&rest” is its pattern
}

— end example]

5 For the purpose of determining whether a parameter pack satisfies a rule regarding entities other than parameter packs, the parameter pack is considered to be the entity that would result from an instantiation of the pattern in which it appears.

6 A parameter pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a parameter pack is only expanded by the innermost enclosing pack expansion. The pattern of a pack expansion shall name one or more parameter packs that are not expanded by a nested pack expansion; such parameter packs are called **unexpanded parameter packs** in the pattern. All of the parameter packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a parameter pack that is not expanded is ill-formed. [Example:

```cpp
template<typename...> struct Tuple {};
```
The instantiation of a pack expansion that is neither a \texttt{sizeof...} expression nor a \texttt{fold-expression} produces a list \( E_1, E_2, \ldots, E_N \), 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:

\begin{enumerate}
  \item if the pack is a template parameter pack, the element is a template parameter (17.1) of the corresponding kind (type or non-type) designating the type or value from the template argument; otherwise,
  \item if the pack is a function parameter pack, the element is an id-expression designating the function parameter that resulted from the instantiation of the pattern where the pack is declared.
\end{enumerate}

All of the \( E_i \) become elements in the enclosing list. [\textit{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. [\textit{Example:}]

\begin{verbatim}

template<class ... T> struct X : T... { };

template<class ... T> struct f<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{verbatim}

The instantiation of a \texttt{sizeof...} expression (8.5.2.3) produces an integral constant containing the number of elements in the parameter pack it expands.

The instantiation of a \texttt{fold-expression} produces:

\begin{enumerate}
  \item \( (E_1 \text{ op } E_2) \text{ op } \cdots \text{ op } E_N \) for a unary left fold,
  \item \( E_1 \text{ op } (\cdots \text{ op } (E_{N-1} \text{ op } E_N)) \) for a unary right fold,
  \item \( ((E \text{ op } E_1) \text{ op } E_2) \text{ op } \cdots \text{ op } E_N \) for a binary left fold, and
  \item \( E_1 \text{ op } (\cdots \text{ op } (E_{N-1} \text{ op } (E_N \text{ op } E))) \) for a binary right fold.
\end{enumerate}

In each case, \( \text{ op } \) is the \texttt{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 \texttt{cast-expression} that did not contain an unexpanded parameter pack. [\textit{Example:}]

\begin{verbatim}

template<typename ...Args> bool all(Args ...args) { return (...) && args; }

\end{verbatim}

\[ 17.6.3 \]
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 14; if the operator is not listed in Table 14, the instantiation is ill-formed.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value when parameter 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>

### 17.6.4 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 \textit{template-id}, the friend declaration refers to a specialization of a function template, otherwise,
2. if the name of the friend is a \textit{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 \textit{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 (17.9.2.6), otherwise,
4. the name shall be an \textit{unqualified-id} that declares (or redeclares) a non-template function.

[Example:

```
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
friend void next_time();
friend void process(task<T>*);
friend task<T>* preempt<T>(task<T>*);
template<class C> friend int func(C);

friend class task<int>;
template<class P> friend class frd;
};
```

Here, each specialization of the \textit{task} class template has the function \textit{next\_time} as a friend; because \textit{process} does not have explicit \textit{template-arguments}, each specialization of the \textit{task} class template has an appropriately typed function \textit{process} as a friend, and this friend is not a function template specialization; because the friend \textit{preempt} has an explicit \textit{template-argument} \( T \), each specialization of the \textit{task} class template has the appropriate specialization of the function template \textit{preempt} as a friend; and each specialization of the \textit{task} class template has all specializations of the function template \textit{func} as friends. Similarly, each specialization of the \textit{task} class template has the class template specialization \textit{task<int>} as a friend, and has all specializations of the class template \textit{frd} as friends. —end example]

2. A friend template may be declared within a class or class template. A friend function template may be defined within a class or class template, but a friend class template may not be defined in a class or class template. In these cases, all specializations of the friend class or friend function template are friends of the class or class template granting friendship. [Example:

```
class A {
    template<class T> friend class B;    // OK
    template<class T> friend void f(T){ /* ... */ }    // OK
};
—end example]

§ 17.6.4 330
A template friend declaration specifies that all specializations of that template, whether they are implicitly instantiated (17.8.1), partially specialized (17.6.5) or explicitly specialized (17.8.3), 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 (17.9.2.5). 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 { }
    struct D {
        void g();
    };
    T h();
    template<T U> T i();
};

template<> struct A<int> {
    struct B { }
    int f();
    struct D {
        void g();
    };
    template<int U> int i();
};

template<> struct A<float*> {
    int *h();
};

class C {
    template<class T> friend struct A<T>::B;
    // grants friendship to A<int>::B even though
    // it is not a specialization of A<T>::B
    template<class T> friend void A<T>::f();
    // does not grant friendship to A<int>::f()
    // because its return type does not match
    // a simple-template-id
    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
    // of those function templates
};
```

--- end example]}

Note: A friend declaration may first declare a member of an enclosing namespace scope (17.7.5). —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 { };
```
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 specifier be used in such a declaration.

A non-template friend declaration shall not have a requires-clause.

### 17.6.5 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 (17.6.5.1). 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.

Each class template partial specialization is a distinct template and definitions shall be provided for the members of a template partial specialization (17.6.5.3).

A class template partial specialization may be constrained (Clause 17). [Example:

```cpp
template<typename T> concept C = true;
template<typename T> struct X { }; // #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 (17.4.4). —end example]

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,

```cpp
template<class T1, class T2, int I> class A { class A<T1, T2, I> { }; // error
—end example]
```

A class template partial specialization may be declared in any scope in which the corresponding primary template may be defined (10.3.1.2, 12.2, 17.6.2). [Example:

```cpp
template<class T> struct A {
    struct C {
        template<class T2> struct B { }; // 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
```

A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type
arguments are specialized.

Within the argument list of a class template partial specialization, the following restrictions apply:

1. The type of a template parameter corresponding to a specialized non-type argument shall not be
dependent on a parameter of the specialization. [Example:

```cpp
template <class T, T t> struct C {};
template <class T> struct C<T, 1>; // error
```

```cpp
template< int X, int (*array_ptr)[X] > class A {};
int array[5];
template< int X > class A<X,&array> { }; // error
```

--- end example ---

2. The specialization shall be more specialized than the primary template (17.6.5.2).

3. The template parameter list of a specialization shall not contain default template argument values.

4. An argument shall not contain an unexpanded parameter pack. If an argument is a pack expansion
(17.6.3), 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]

## 17.6.5.1 Matching of class template partial specializations [temp.class.spec.match]

When a class template is used in a context that requires an instantiation of the class, it is necessary to
determine whether the instantiation is to be generated using the primary template or one of the partial
specializations. This is done by matching the template arguments of the class template specialization with
the template argument lists of the partial specializations.

1. If exactly one matching specialization is found, the instantiation is generated from that specialization.

2. If more than one matching specialization is found, the partial order rules (17.6.5.2) 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 (17.9.2), and the deduced template arguments satisfy the associated constraints of the partial specialization, if any (17.4.2). [Example:

```cpp
template<class T1, class T2, int I> class A { }; // #1
template<class T, int I> class A<T, T*, I> { }; // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { }; // #3
template<class T> class A<int, T*, 5> { }; // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { }; // #5
```

A<int, int, 1> a1; // uses #1
A<int, int*, 1> a2; // uses #2, T is int, I is 1
A<int, char*, 5> a3; // uses #4, T is char
A<int, char*, 1> a4; // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5; // ambiguous: matches #3 and #5

— end example] [Example:

```cpp
template<typename T> concept C = requires (T t) { t.f(); };

template<typename T> struct S { }; // #1

template<C T> struct S<T> { }; // #2
```

```cpp
struct Arg { void f(); }; // #2
```

S<int> s1; // uses #1; the constraints of #2 are not satisfied
S<Arg> s2; // uses #2; both constraints are satisfied but #2 is more specialized

— 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:

```cpp
template <int I, int J, class T> class X { }; // error

template <int I> class X<I, I, int> { }; // error

template <int I, int J, int K> struct B { };

template <int I> struct B<I, I*2, 2> { }; // OK
```

— end example]

In a type name that refers to a class template specialization, (e.g., `A<int, int, 1>`) the argument list shall match the template parameter list of the primary template. The template arguments of a specialization are deduced from the arguments of the primary template.

### 17.6.5.2 Partial ordering of class template specializations

For two class template partial specializations, the first is more specialized than the second if, given the following rewrite to two function templates, the first function template is more specialized than the second according to the ordering rules for function templates (17.6.6.2):

1. Each of the two function templates has the same template parameters and associated constraints (17.4.2) as the corresponding partial specialization.
2. Each function template has a single function parameter whose type is a class template specialization where the template arguments are the corresponding template parameters from the function template for each template argument in the template-parameter-list of the simple-template-id of the partial specialization.

[Example:

```cpp
template<int I, int J, class T> class X { }; // #1

template<int I> class X<I, I, int> { }; // #2

template<int I0, int J0> void f(X<I0, J0, int>); // A

template<int I0> void f(X<I0, I0, int>); // B
```

§ 17.6.5.2 334
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> concept S { };  
template<int T> class S<T> { };  // #1  
template<int T> void S<T>::f() { }  // #2  

template<int T> void S<T>::g() { }  // A  
template<int T> void S<T>::h() { }  // B  
```

The partial specialization #2 is more specialized than #1 because $B$ is more specialized than $A$. —end example]

### 17.6.5.3 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 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>::g() { }  

// explicit specialization  
template<> void A<char,2>::h() { }  
```

int main() {
  A<char,0> a0;  
  A<char,2> a2;  
  a0.f();    // OK, uses definition of primary template's member  
a2.g();    // OK, uses definition of partial specialization's member  
a2.h();    // OK, uses definition of explicit specialization's member  
a2.f();    // ill-formed, no definition of f for A<T,2>; the primary template is not used here  
}
2 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 (17.8.1, 17.8.2), 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:

```
template<class T> struct A {
    template<class T2> struct B {};
    template<class T2> struct B<T2*> {};
};
```

```
template<> template<class T2> struct A<short>::B {};
```

```
A<char>::B<int*> abcip;
// uses #2
A<short>::B<int*> absip;
// uses #3
A<char>::B<int> abci;
// uses #1
```

—end example—

17.6.6 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:

```
template<class T> class Array { }
```

```
template<class T> void sort(Array<T>&);
```

—end example—

2 A function template can be overloaded with other function templates and with non-template functions (11.3.5). 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.\[142\]

17.6.6.1 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:

```
// translation unit 1:
template<class T>
void f(T*);
void g(int* p) {
    f(p); // calls f<int*>(int*)
}
```

```
// translation unit 2:
template<class T>
void f(T);
void h(int* p) {
    f(p); // calls f<int*>(int*)
}
```

—end example—

2 Such specializations are distinct functions and do not violate the one-definition rule (6.2).

3 The signature of a function template is defined in Clause 3. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature. [Note: Two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

```
template<class T> void f();
template<int I> void f(); // OK: overloads the first template
// distinguishable with an explicit template argument list
```

—end note—

[142] 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.
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:]

```
template <int I, int J> A<I+J> f(A<I>, A<J>); // #1
template <int K, int L> A<K+L> f(A<K>, A<L>); // same as #1
template <int I, int J> A<I-J> f(A<I>, A<J>); // different from #1
```

—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-expressions are never considered equivalent. [Note: The intent is to avoid lambda-expressions appearing in the signature of a function template with external linkage. —end note] For determining whether two dependent names (17.7.2) 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:]

```
template <int I, int J> void f(A<I+J>); // #1
```

```
template <int K, int L> void f(A<K+L>); // same as #1
```

```
template <class T> decltype(g(T())) h();
```

```
template <class T> decltype(g(T())) h() // redeclaration of h() uses the earlier lookup...
{
    return g(T());
}
```

```
int i = h<int>(); // template argument substitution fails; g(int)
```

// ill-formed, no diagnostic required: the two expressions are functionally equivalent but not equivalent
```
template <int N> void foo(const char (*s)[]){(N)}, N));
template <int N> void foo(const char (*s)[]){(N)}, N));
```

// two different declarations because the non-dependent portions are not considered equivalent
```
template <class T> void spam(decltype([]){}) (s)[sizeof(T)]);
```

```
template <class T> void spam(decltype([]){}) (s)[sizeof(T)]);
```

—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-heads are equivalent if their template-parameter-lists 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:

(6.1) — they declare template parameters of the same kind,

(6.2) — if either declares a template parameter pack, they both do,

(6.3) — if they declare non-type template parameters, they have equivalent types,

(6.4) — if they declare template template parameters, their template parameters are equivalent, and

(6.5) — 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 (17.4.1) 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]
```

### 17.6.6.2 Partial ordering of function templates

If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (17.9.2) 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. During overload resolution for a call to a function template specialization (16.3.3);
2. When the address of a function template specialization is taken;
3. When a placement operator delete that is a function template specialization is selected to match a placement operator new (6.6.4.4.2, 8.5.2.4);
4. When a friend function declaration (17.6.4), an explicit instantiation (17.8.2) or an explicit specialization (17.8.3) 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 17.4.4.

To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs (17.6.3) 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 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 “lvalue reference to \( cv \ A \)”.

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
```

§ 17.6.6.2
// 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 17.9.2.4.

[ 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() {
        const int* p;
        f(p);           // f(const T*) is more specialized than f(T) or f(T*)
        float x;
        g(x);          // ambiguous: g(T) or g(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
    template<class T, class... U> void g(T, U ...);    // #4
    void h() {
        f<int>(42, (A<int, int*>*0));       // calls #2
        f<int>(42);                        // error: ambiguous
    }
g(42);                          // error: ambiguous

—end example] [Example:

template<class T, class... U> void f(T, U...);      // #1

template<class T>   void f(T);                        // #2

template<class T, class... U> void g(T*, U...);     // #3

template<class T>   void g(T);                        // #4

void h(int i) {
  f(&i);                                      // error: ambiguous
  g(&i);                                      // OK: calls #3
}

—end example] —end note]

17.6.7 Alias templates [temp.alias]

1 A template-declaration in which the declaration is an alias-declaration (Clause 10) 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>
    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> 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

§ 17.6.7 340
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.  

```cpp
template <class T>
using A = decltype([] { });  // A<int> and A<char> refer to different closure types
```

### 17.6.8 Concept definitions

1. A **concept** is a template that defines constraints on its template arguments.
2. A **concept-definition** declares a concept. Its **identifier** becomes a **concept-name** referring to that concept within its scope.  

```cpp
template<typename T>
concept C = requires(T x) {
    { x == x } -> bool;
};
```

```cpp
template< typename T >
    requires C<T>  // C constrains f1(T) in constraint-expression
T f1(T x) { return x; }
```

```cpp
template<C T>
    // C constrains f2(T) as a constrained-parameter
T f2(T x) { return x; }
```

### 17.7 Name resolution

1. 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** (17.7.2).
   (1.3) — Names from scopes which are visible within the template definition.
2. A name used in a template declaration or definition and that is dependent on a **template-parameter** is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword **typename**.  

```cpp
// no B declared here

class X;

template<class T> class Y {
    class Z;  // forward declaration of member class
    
    void f() {
        X* a1;  // declare pointer to X
        T* a2;  // declare pointer to T
        Y<T>* a3;  // declare pointer to Y<T>
        Z* a4;  // declare pointer to Z
        typedef typename T::A TA;
        TA* a5;  // declare pointer to T's A
        typename T::A* a6;  // declare pointer to T's A
        T::A* a7;  // T::A is not a type name:
    }
```

§ 17.7 341
When a qualified-id is intended to refer to a type that is not a member of the current instantiation (17.7.2.1) and its nested-name-specifier refers to a dependent type, it shall be prefixed by the keyword typename, forming a typename-specifier. If the qualified-id in a typename-specifier does not denote a type or a class template, the program is ill-formed.

**typename-specifier:**

```
typename nested-name-specifier identifier
typename nested-name-specifier template_opt simple-template-id
```

If a specialization of a template is instantiated for a set of template-arguments such that the qualified-id prefixed by typename does not denote a type or a class template, the specialization is ill-formed. The usual qualified name lookup (6.4.3) is used to find the qualified-id even in the presence of typename.

```
struct A {
    struct X {
    }
    int X;
};
struct B {
    struct X {
    }
};
template<class T> void f(T t) {
    typename T::X x;
}
void foo() {
    A a;
    B b;
    f(b);  // OK: T::X refers to B::X
    f(a);  // error: T::X refers to the data member A::X not the struct A::X
}
```

A qualified name used as the name in a class-or-decltype (Clause 13) 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]

If, for a given set of template arguments, a specialization of a template is instantiated that refers to a qualified-id that denotes a type or a class template, and the qualified-id refers to a member of an unknown specialization, the qualified-id shall either be prefixed by typename or shall be used in a context in which it implicitly names a type as described above. [Example:

```
template <class T> void f(int i) {
    T::x * i;  // T::x must not be a type
}
struct Foo {
    typedef int x;
};
struct Bar {
    static int const x = 5;
};
int main() {
    f<Bar>(1);  // OK
    f<Foo>(1);  // error: Foo::x is a type
}
```
Within the definition of a class template or within the definition of a member of a class template following the declarator-id, the keyword typename is not required when referring to the name of a previously declared member of the class template that declares a type or a class template. [Note: Such names can be found using unqualified name lookup (6.4.1), class member lookup (6.4.3.1) into the current instantiation (17.7.2.1), or class member access expression lookup (6.4.5) when the type of the object expression is the current instantiation (17.7.2.2). —end note] [Example:

```cpp
template<class T> struct A {
  typedef int B;
  B b; // OK, no typename required
};
```
—end example]

Knowing which names are type names allows the syntax of every template to be checked. The program is ill-formed, no diagnostic required, if:

1. no valid specialization can be generated for a template or a substatement of a constexpr if statement (9.4.1) within a template and the template is not instantiated, or
2. no substitution of template arguments into a partial-concept-id or requires-clause would result in a valid expression, or
3. every valid specialization of a variadic template requires an empty template parameter pack, or
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
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 (8.6) 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
  }
};

§ 17.7 343
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 (17.7.2). [Example:

```c++
#include <iostream>
using namespace std;

template<class T> class Set {
    T* p;
    int cnt;
public:
    Set();
    Set<T>(const Set<T>&);
    void printall() {
        for (int i = 0; i<cnt; i++)
            cout << p[i] << '\n';
    }
};
```

in the example, `i` is the local variable declared in `printall`, `cnt` is the member declared in `Set`, and `cout` is the standard output stream declared in `<iostream>`. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual template-arguments are known. For example, even though the name `operator<<` is known within the definition of `printall()` and a declaration of it can be found in `<iostream>`, the actual declaration of `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (17.7.2). —end example]

If a name does not depend on a template-parameter (as defined in 17.7.2), 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:

```c++
void f(char);

template<class T> void g(T t) {
    f(1);          // f(char)
    f(T(1));       // dependent
    f(t);          // dependent
    dd++;          // not dependent; error: declaration for dd not found
}

enum E { e };
void f(E);

double dd;
void h() {
    g(e);          // will cause one call of f(char) followed by two calls of f(E)
    g('a');        // will cause three calls of f(char)
}
END EXAMPLE]

[Note: For purposes of name lookup, default arguments and `noexcept-specifiers` of function templates and default arguments and `noexcept-specifiers` of member functions of class templates are considered definitions (17.6). —end note]
17.7.1 Locally declared names

Like normal (non-template) classes, class templates have an injected-class-name (Clause 12). The injected-class-name can be used as a template-name or a type-name. When it is used with a template-argument-list, as a template-argument for a template template-parameter, or as the final identifier in the elaborated-type-specifier of a friend class template declaration, it refers to the class template itself. Otherwise, it is equivalent to the template-name followed by the template-parameters of the class template enclosed in <>

Within the scope of a class template specialization or partial specialization, when the injected-class-name is used as a type-name, it is equivalent to the template-name followed by the template-arguments of the class template specialization or partial specialization enclosed in <>

The injected-class-name of a class template or class template specialization can be used either as a template-name or a type-name wherever it is in scope.

A lookup that finds an injected-class-name (13.2) 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.

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.

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.
template<class T, int i> class Y {
    int T; // error: template-parameter redeclared
    void f() {
        char T; // error: template-parameter redeclared
    }
};

template<class X> class X; // error: template-parameter redeclared

— end example

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:

    template<class T> struct A {
        struct B { /* ... */ };  
        typedef void C;
        void f();
        template<class U> void g(U);
    };

    template<class B> void A<B>::f() {
        B b; // A's B, not the template parameter
    }

    template<class B> template<class C> void A<B>::g(C) {
        B b; // A's B, not the template parameter
        C c; // the template parameter C, not A's C
    }

    — end example
]

8 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:

    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
]

9 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 (17.7.2.1), 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:

    struct A {
        struct B { /* ... */ };
        int a;
        int Y;
    };

    template<class B, class a> struct X : A {
        B b; // A's B
        a b; // error: A's a isn't a type name
    };

    — end example
]
17.7.2 Dependent names

Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct depends on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. An expressions 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 (8.6) may depend on a template parameter) as described in this subclause. 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

(1.1) — any of the expressions in the expression-list is a pack expansion (17.6.3),

(1.2) — any of the expressions or braced-init-lists in the expression-list is type-dependent (17.7.2.2), or

(1.3) — 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. Such names are unbound and are looked up at the point of the template instantiation (17.7.4.1) in both the context of the template definition and the context of the point of instantiation.

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.

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>.

Example:

```cpp
struct A {
    struct B { /* ... */ };  
    int a;
    int Y;
};

int a;

template<class T> struct Y : T {
    struct B { /* ... */ };  
    B b;                      // The B defined in Y
    void f(int i) { a = i; }  // ::a
    Y* p;                     // Y<T>
};

Y<A> ya;
```

§ 17.7.2
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 —

### 17.7.2.1 Dependent types

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 12) 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 \(n\)th template parameter is a parameter pack, the \(n\)th template argument is a pack expansion (17.6.3) whose pattern is the name of the parameter pack.

2 The template argument list of a primary template is a template argument list in which the \(n\)th template argument has the value of the \(n\)th template parameter of the class template. If the \(n\)th template parameter is a template parameter pack (17.6.3), the \(n\)th template argument is a pack expansion (17.6.3) 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>::::B* p3; // A<T>::::B is not the current instantiation
    };
};

template <class T> class A<T*> {
    A<T>** p1;          // A<T> is the current instantiation
    A<T>* p2;           // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
    B<T1, T2, I>> b1;    // refers to the current instantiation
    B<T2, T1, I> b2;     // not the current instantiation

typedef T1 my_T1;
static const int my_I = I;
static const int my_I2 = I+0;
static const int my_I3 = my_I;
static const long my_I4 = I;
static const int my_I5 = (I);
B<my_T1, T2, my_I>> b3;  // refers to the current instantiation
}
```
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]

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 (8.5.1.5) 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:

```cpp
template <class T> class A {
    static const int i = 5;
    int n1[i];
    int n2[A::i]; // A::i refers to a member of the current instantiation
    int n3[A<T>::i]; // A<T>::i refers to a member of the current instantiation
    int f();
};

template <class T> int A<T>::f() {
    return i; // i refers to a member of the current instantiation
}
```

— end example]

A name is a dependent member of the current instantiation if it is a member of the current instantiation that, when looked up, refers to at least one member of a class that is the current instantiation.

A name is a member of an unknown specialization if it is

(6.1) — A qualified-id in which the nested-name-specifier names a dependent type that is not the current instantiation.

(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.
An id-expression denoting the member in a class member access expression (8.5.1.5) in which either

- the type of the object expression is the current instantiation, the current instantiation has at least one dependent base class, and name lookup of the id-expression does not find a member of a class that is the current instantiation or a non-dependent base class thereof; or

- the type of the object expression is dependent and is not the current instantiation.

If a qualified-id in which the nested-name-specifier refers to the current instantiation is not a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the qualified-id is not instantiated; no diagnostic required. Similarly, if the id-expression in a class member access expression for which the type of the object expression is the current instantiation does not refer to a member of the current instantiation or a member of an unknown specialization, the program is ill-formed even if the template containing the member access expression is not instantiated; no diagnostic required. [Example:

```cpp
template<class T> class A {
  typedef int type;
  void f() {
    A<T>::type i; // OK: refers to a member of the current instantiation
    typename A<T>::other j; // error: neither a member of the current instantiation nor a member of an unknown specialization
  }
};
```
—end example]

8 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 12.2.2

—end example]

9 A type is dependent if it is

- a template parameter,
- a member of an unknown specialization,
- a nested class or enumeration that is a dependent member of the current instantiation,
- a cv-qualified type where the cv-unqualified type is dependent,
- a compound type constructed from any dependent type,
- an array type whose element type is dependent or whose bound (if any) is value-dependent,
- a function type whose exception specification is value-dependent,
- 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 12) of a class template used without a template-argument-list. — end note] , or

(9.9) — denoted by decltype(expression), where expression is type-dependent (17.7.2.2).

[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]

17.7.2.2 Type-dependent expressions

Except as described below, an expression is type-dependent if any subexpression is type-dependent.

this is type-dependent if the class type of the enclosing member function is dependent (17.7.2.1).

An id-expression is type-dependent if it contains

(3.1) — an identifier associated by name lookup with one or more declarations declared with a dependent type,

(3.2) — an identifier associated by name lookup with a non-type template-parameter declared with a type that contains a placeholder type (10.1.7.4),

(3.3) — 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,

(3.4) — an identifier associated by name lookup with a structured binding declaration (11.5) whose brace-or-equal-initializer is type-dependent,

(3.5) — the identifier __func__ (11.4.1), where any enclosing function is a template, a member of a class template, or a generic lambda,

(3.6) — a template-id that is dependent,

(3.7) — a conversion-function-id that specifies a dependent type, or

(3.8) — a nested-name-specifier or a qualified-id that names a member of an unknown specialization;

or if it names a dependent member of the current instantiation that is a static data member of type “array of unknown bound of T” for some T (17.6.1.3). Expressions of the following forms are type-dependent only if the type specified by the type-id, simple-type-specifier or new-type-id is dependent, even if any subexpression is type-dependent:

\[
\begin{align*}
\text{simple-type-specifier} & \ (\text{expression-list}_{\text{opt}}) \\
\text{::opt new} & \ (\text{new-placement}_{\text{opt}} \ \text{new-type-id} \ \text{new-initializer}_{\text{opt}}) \\
\text{::opt new} & \ (\text{new-placement}_{\text{opt}} \ (\text{type-id}) \ \text{new-initializer}_{\text{opt}}) \\
\text{dynamic_cast} < & \ (\text{type-id}) \ (\text{expression}) \\
\text{static_cast} < & \ (\text{type-id}) \ (\text{expression}) \\
\text{const_cast} < & \ (\text{type-id}) \ (\text{expression}) \\
\text{reinterpret_cast} < & \ (\text{type-id}) \ (\text{expression}) \\
& \ (\text{type-id}) \ \text{cast-expression}
\end{align*}
\]

Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

\[
\begin{align*}
\text{literal} & \\
\text{postfix-expression} & . \ \text{pseudo-destructor-name} \\
\text{postfix-expression} & \rightarrow \ \text{pseudo-destructor-name} \\
\text{sizeof} & \ \text{unary-expression} \\
\text{sizeof} & \ (\text{type-id}) \\
\text{sizeof} & \ldots \ (\text{identifier}) \\
\text{alignof} & \ (\text{type-id}) \\
\text{typeid} & \ (\text{expression}) \\
\text{typeid} & \ (\text{type-id}) \\
\text{::opt delete} & \ \text{cast-expression} \\
\text{::opt delete} & [\ ] \ \text{cast-expression} \\
\text{throw} & \ \text{assignment-expression}_{\text{opt}} \\
\text{noexcept} & \ (\text{expression})
\end{align*}
\]

[Note: For the standard library macro offsetof, see 21.2. — end note]

A class member access expression (8.5.1.5) is type-dependent if the expression refers to a member of the current instantiation and the type of the referenced member is dependent, or the class member access expression refers to a member of an unknown specialization. [Note: In an expression of the form x.y or
xp->y the type of the expression is usually the type of the member y of the class of x (or the class pointed to by xp). However, if x or xp refers to a dependent type that is not the current instantiation, the type of y is always dependent. If x or xp refers to a non-dependent type or refers to the current instantiation, the type of y is the type of the class member access expression. — end note]

A braced-init-list is type-dependent if any element is type-dependent or is a pack expansion.

A fold-expression is type-dependent.

17.7.2.3 Value-dependent expressions

Except as described below, an expression used in a context where a constant expression is required is value-dependent if any subexpression is value-dependent.

An id-expression is value-dependent if:

1. it is type-dependent,
2. it is the name of a non-type template parameter,
3. it names a static data member that is a dependent member of the current instantiation and is not initialized in a member-declarator,
4. it names a static member function that is a dependent member of the current instantiation, or
5. it is a constant with literal type and is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the unary-expression or expression is type-dependent or the type-id is dependent:

- sizeof unary-expression
- sizeof ( type-id )
- typeid ( expression )
- typeid ( type-id )
- alignof ( type-id )
- noexcept ( expression )

[ Note: For the standard library macro offsetof, see 21.2. — 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-listopt )
- 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 (8.6) 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.

17.7.2.4 Dependent template arguments

A type template-argument is dependent if the type it specifies is dependent.

A non-type template-argument is dependent if its type is dependent or the constant expression it specifies is value-dependent.

Furthermore, a non-type template-argument is dependent if the corresponding non-type template-parameter is of reference or pointer type and the template-argument designates or points to a member of the current instantiation or a member of a dependent type.

A template template-argument is dependent if it names a template-parameter or is a qualified-id that refers to a member of an unknown specialization.
17.7.3 Non-dependent names  
Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [Example:

```cpp
void g(double);
void h();

template<class T> class Z {
public:
    void f() {
        g(1);        // calls g(double)
        h++;         // ill-formed: cannot increment function; this could be diagnosed
    }
};
void g(int);   // not in scope at the point of the template definition, not considered for the call g(1)
```
—end example]

17.7.4 Dependent name resolution  
In resolving dependent names, names from the following sources are considered:

1. Declarations that are visible at the point of definition of the template.
2. Declarations from namespaces associated with the types of the function arguments both from the instantiation context (17.7.4.1) and from the definition context.

17.7.4.1 Point of instantiation  
For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

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. The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.
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, for any such specialization that has a point of instantiation within the translation unit, 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.

17.7.4.2 Candidate functions

For a function call where the postfix-expression is a dependent name, the candidate functions are found using the usual lookup rules (6.4.1, 6.4.2) except that:

1. For the part of the lookup using unqualified name lookup (6.4.1), only function declarations from the template definition context are found.

2. For the part of the lookup using associated namespaces (6.4.2), only function declarations found in either the template definition context or the template instantiation context are found.

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.

17.7.5 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 (14.3). Such names may be found under the rules for associated classes (6.4.2).[143] [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
}
```

—end example]

17.8 Template instantiation and specialization

The act of instantiating a function, 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.

An explicit specialization may be declared for a function 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 class template, a member of a class template or a class member template, the name of the class that is explicitly specialized shall be a simple-template-id. In the explicit

---

143) Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.
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:

```cpp
template<class T = int> struct A {
    static int x;
};
template<class U> void g(U) {}

template<> struct A<double> { }; // specialize for T == double
template<> struct A<> { }; // specialize for T == int
template<> void g<char> () {}; // specialize for U == char

template<> int A<char>::x = 0; // specialize for T == char
```

—end example]

4 An instantiated template specialization can be either implicitly instantiated (17.8.1) for a given argument list or be explicitly instantiated (17.8.2). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (17.8.3).

5 For a given template and a given set of template-arguments,

(5.1) — an explicit instantiation definition shall appear at most once in a program,
(5.2) — an explicit specialization shall be defined at most once in a program (according to 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.

6 The usual access checking rules do not apply to names in a declaration of an explicit instantiation or explicit specialization, with the exception of names appearing in a function body, default argument, base-clause, member-specification, enumerator-list, or static data member or variable template initializer. [Note: 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]

7 Each class template specialization instantiated from a template has its own copy of any static members. [Example:

```cpp
template<class T> class X {
    static T s;
};
template<class T> T X<T>::s = 0;
X<int> a;
X<char*> b;
```

X<int> has a static member s of type int and X<char*> has a static member s of type char*. —end example]

8 If a function declaration acquired its function type through a dependent type (17.7.2.1) without using the syntactic form of a function declarator, the program is ill-formed. [Example:

```cpp
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]
17.8.1 Implicit instantiation

Unless a class template specialization has been explicitly instantiated (17.8.2) or explicitly specialized (17.8.3), 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:

```cpp
template<class T> class B { /* ... */ };  
template<class T> class D : public B<T> { /* ... */ };  
void f(void*);  
void f(B<int>*);  

void g(D<int>* p, D<char>* pp, D<double>* ppp) {  
    f(p); // instantiation of D<int> required: call f(B<int>*)  
    B<char>* q = pp; // instantiation of D<char> required: convert D<char>* to B<char>*  
    delete ppp; // instantiation of D<double> required  
}
```

—end example]

If a class template has been declared, but not defined, at the point of instantiation (17.7.4.1), the instantiation yields an incomplete class type (6.7). [Example:

```cpp
template<class T> class X;  
X<char> ch; // error: incomplete type X<char>  
```

—end example] [Note: Within a template declaration, a local class (12.4) 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 partial-concept-ids or requires-clauses). As a result, the dependent names are looked up, the semantic constraints are checked, and any templates used are instantiated as part of the instantiation of the entity within which the local class or enumeration is declared. —end note]

The implicit instantiation of a class template specialization causes the implicit instantiation of the declarations, but not of the definitions, default arguments, or noexcept-specifiers of the class member functions, member classes, scoped member enumerations, static data members, member templates, and friends; and it causes the implicit instantiation of the definitions of unscoped member enumerations and member anonymous unions. However, for the purpose of determining whether an instantiated redeclaration is valid according to 6.2 and 12.2, a declaration that corresponds to a definition in the template is considered to be a definition. [Example:

```cpp
template<class T, class U>  
struct Outer {  
    template<class X, class Y> struct Inner;  
    template<class Y> struct Inner<T, Y>; // #1a  
    template<class Y> struct Inner<T, Y> { }; // #1b; OK: valid redeclaration of #1a  
    template<class Y> struct Inner<U, Y> { }; // #2  
};  

Outer<int, int> outer; // error at #2  
```

Outer<int, int>::Inner<int, Y> is redeclared at #1b. (It is not defined but noted as being associated with a definition in Outer<T, U>.) #2 is also a redeclaration of #1a. It is noted as associated with a definition, so it is an invalid redeclaration of the same partial specialization.

```cpp
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]

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

§ 17.8.1 356
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 (8.6), 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 (16.3) 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);
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 (17.9.3).

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 (9.4.1), 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 specializations in the default argument to be implicitly instantiated.

Implicitly instantiated class, function, and variable template specializations are placed in the namespace where the template is defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the enclosing class or class template is defined. [Example:

```cpp
namespace N {
  template<class T> class List {
    public:
      T* get();
    }
  
  template<class K, class V> class Map {
    public:
      N::List<V> lt;
      V get(K);
    }

    void g(Map<const char*,int>& m) {
      int i = m.get("Nicholas");
    }
  
  a call of lt.get() from Map<const char*,int>::get() would place List<int>::get() in the namespace N rather than in the global namespace. —end example]
```

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 (8.4.5.1) – 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 \( f \).

Each default argument is instantiated independently. [Example:

```cpp
template<class T> void f(T x, T y = ydef(T()), T z = zdef(T()));

class A {
};

A zdef(A);
```
```cpp
void g(A a, A b, A c) {
    f(a, b, c); // no default argument instantiation
    f(a, b);   // default argument z = zdef(T()) instantiated
    f(a);     // ill-formed; ydef is not declared
}

The noexcept-specifier of a function template specialization is not instantiated along with the function declaration; it is instantiated when needed (18.4). 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: 17.7.4.1 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 partial-concept-ids 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 17.4.2 and 17.4.1.2 when determining whether the constraints are satisfied. [Note: The satisfaction of constraints is determined during name lookup or overload resolution (16.3). —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
};

S<char> s1;         // error: no matching constructor
S<char[8]> s2;      // OK, calls #2
```

When S<char> is instantiated, both constructors are part of the specialization. Their constraints are not satisfied, and they suppress the implicit declaration of a default constructor for S<char> (15.1), so there is no viable constructor for s1. —end example]  

[Example:

```cpp
template<typename T> struct S1 {
    template<typename U>
    requires false
    struct Inner1;       // ill-formed, no diagnostic required
};
template<typename T> struct S2 {
    template<typename U>
    requires (sizeof(T[-(int)sizeof(T)]) > 1)
    struct Inner2;       // ill-formed, no diagnostic required
};
```

The class S1<T>::Inner1 is ill-formed, no diagnostic required, because it has no valid specializations. S2 is ill-formed, no diagnostic required, since no substitution into the constraints of its Inner2 template would result in a valid expression. —end example]

§ 17.8.1
17.8.2 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 or constexpr specifiers.

The syntax for explicit instantiation is:

```
eextern 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.

If the explicit instantiation is for a class or member class, the elaborated-type-specifier in the declaration shall include a simple-template-id; otherwise, the declaration shall be a simple-declaration whose init-declarator-list comprises a single init-declarator that does not have an initializer. If the explicit instantiation is for a function or member function, the unqualified-id in the declarator shall be either a template-id or, where all template arguments can be deduced, a template-name or operator-function-id. [Note: The declaration may declare a qualified-id, in which case the unqualified-id of the qualified-id must be a template-id. — end note] If the explicit instantiation is for a function or 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 (10.3.1), any namespace from its enclosing namespace set. [Note: Regarding qualified names in declarations, see 11.3. — end note] [Example:

```
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&);  
```

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 (Clause 15), the program is ill-formed.

The declaration in an explicit-instantiation and the declaration produced by the corresponding substitution into the templated function, variable, or class are two declarations of the same entity. [Note: These declarations are required to have matching types as specified in 6.5, except as specified in 18.4. —end example:

```
template<typename T> T var = {};  
template float var<float>;  // OK, instantiated variable has type float  
template int var<int[16]>[];  // OK, absence of major array bound is permitted  
template int *var<int>;  // error: instantiated variable has type int  

template<typename T> auto av = T();  
template int av<int>;  // OK, variable with type int can be redeclared with type auto  

template<typename T> auto f() {}  
template void f<int>();  // error: function with deduced return type  // redeclared with non-deduced return type (10.1.7.4)
```
Despite its syntactic form, the _declaration_ in an _explicit-instantiation_ for a variable is not itself a definition and does not conflict with the definition instantiated by an explicit instantiation definition for that variable.

For a given set of template arguments, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, for an explicit instantiation definition the definition of a function template, a variable template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

An explicit instantiation of a class, function template, or variable template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [Example:

```cpp
namespace N {
    template<class T> class Y { void mf() { });
}

using N::Y;
template class Y<int>;
// error: class template Y not visible in the global namespace

// 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 (17.9.2). [Example:

```cpp
template<class T> class Array { /* ... */ }

// instantiate sort(Array<int>&)
template void sort<>(Array<int>&);

--- end example ---
```

An explicit instantiation of a constrained template shall satisfy that template’s associated constraints (17.4.2). The satisfaction of constraints is determined when forming the template name of an explicit instantiation in which all template arguments are specified (17.2), or, for explicit instantiations of function templates, during template argument deduction (17.9.2.6) when one or more trailing template arguments are left unspecified. [Note: An explicit instantiation of certain implementation-dependent data about the class. --- 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 (17.4.2, 17.4.1), 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 (10.1.7.4), _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 (17.8.1) in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required. [Note: This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. —end note] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [Example:

```c
char* p = 0;
template<class T> T g(T x = &p) { return x; }
template int g<int>(int); // OK even though &p isn’t an int.
```

—end example]

### 17.8.3 Explicit specialization

An explicit specialization of any of the following:

1. function template
2. class template
3. variable template
4. member function of a class template
5. static data member of a class template
6. member class of a class template
7. member enumeration of a class template
8. member class template of a class or class template
9. member function template of a class or class template

can be declared by a declaration introduced by `template<>`; that is:

```
extricplicit-specialization:
template <> declaration
```

[Example:

```c
template<class T> class stream;
template<> class stream<char> { /* ... */};
```

```c
template<class T> class Array { /* ... */};
template<class T> void sort(Array<T>& v) { /* ... */}
```

```c
template<> void sort<char*>(Array<char*>&);
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`; other streams will be handled by class template specializations instantiated from the class template. Similarly, `sort<char*>` will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template. —end example]

An explicit specialization may be declared in any scope in which the corresponding primary template may be defined (10.3.1.2, 12.2, 17.6.2).

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:

```c
template<> class X<int> { /* ... */}; // error: X not a template
```

template<class T> class X;

§ 17.8.3
template<> class X<char*> { /* ... */ }; // OK: X is a template
— end example]

A member function, a member function template, a member class, a member enumeration, a member class template, a static data member, or a static data member template of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; in this case, the definition of the class template shall precede the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (Clause 15), 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:

template<class T> struct A {
  struct B { }; // OK: B is a template
  template<class U> struct C { }; // OK: C is a template
};

template<> struct A<int> {
  void f(int);
};

void h() {
  A<int> a;
  a.f(16); // A<int>::f must be defined somewhere
}

// template<> not used for a member of an explicitly specialized class template
void A<int>::f(int) { /* ... */ }

template<> struct A<char>::B { // OK: B is a template
  void f();
}; // template<> also not used when defining a member of an explicitly specialized member class
void A<char>::B::f() { /* ... */ }

template<> template<class U> struct A<char>::C { // OK: C is a template
  void f();
}; // template<> is used when defining a member of an explicitly specialized member class template
// specialized as a class template
template<> void A<char>::C<U>::f() { /* ... */ }

template<> struct A<short>::B { // OK: B is a template
  void f();
};

template<> void A<short>::B::f() { /* ... */ } // error: template<> not permitted

template<> template<class U> struct A<short>::C { // OK: C is a template
  void f();
};

template<> template<class U> void A<short>::C<U>::f() { /* ... */ } // error: template<> required
— end example]

If a template, a member template or a member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit
instantiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. If
the program does not provide a definition for an explicit specialization and either the specialization is used in
a way that would cause an implicit instantiation to take place or the member is a virtual member function,
the program is ill-formed, no diagnostic required. An implicit instantiation is never generated for an explicit
specialization that is declared but not defined. [Example:

```c++
  class String {};
  template<class T> class Array { /* ... */ };  
  template<class T> void sort(Array<T>& v) { /* ... */ }

  void f(Array<String>& v) {
    sort(v);  // use primary template sort(Array<T>&), T is String
  }

  template<> void sort<String>(Array<String>& v);  // error: specialization after use of primary template
  template<> void sort<char*>(Array<char*>& v);    // OK: sort<char*> not yet used

  template<class T> struct A {
    enum E : T;
    enum class S : T;
  };
  template<> enum A<int>::E : int { eint };  // OK
  template<> enum class A<int>::S : int { sint }; // OK

  template<class T> enum A<T>::E : T { eT };
  template<class T> enum class A<T>::S : T { sT };
  template<> enum A<char>::E : char { echar };  // ill-formed, A<char>::E was instantiated
  template<> enum class A<char>::S : char { schar }; // 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, member class
templates of class templates, 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:

```c++
namespace N {
  template<class T> class X { /* ... */ };
  template<class T> class Y { /* ... */ };

  template<> class X<int> { /* ... */ };  // OK: specialization in same namespace
  template<> class Y<double>;           // forward-declare intent to specialize for double

  template<> class N::Y<double> { /* ... */ }; // OK: specialization in enclosing namespace
  template<> class N::Y<short> { /* ... */ }; // OK: specialization in enclosing namespace

  —end example
```

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:

```c++
  template<class T> class X;        // X is a class template
  template<> class X<int>;

  X<int>* p;     // OK: pointer to declared class X<int>
  X<int> x;      // error: object of incomplete class X<int>
```

§ 17.8.3 364
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 (17.4.2). The satisfaction of constraints is determined when forming the template name of an explicit specialization in which all template arguments are specified (17.2), or, for explicit specializations of function templates, during template argument deduction (17.9.2.6) 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 (17.6.6).

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) { /* ... */ } // OK: inline
template<> int g<>(int) { /* ... */ } // OK: not inline
```
—end example]

An explicit specialization of a static data member of a template or an explicit specialization of a static data member template is a definition if the declaration includes an initializer; otherwise, it is a declaration. [Note: 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);

// member template specialization
template<> template<>
    void A<int>::g1(int, char); // X1 deduced as char
```
template<> template<> void A<int>::g2<char>(int, char); // X2 specified as char

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }

— end example ]

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:

    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:

    template <class T1> class A {
        template<class T2> class B {
            template<class T3> void mf1(T3);
            void mf2();
        };
    };

    template <> template <class X> class A<int>::B {
        template <class T> void mf1(T);
    };

    template <> template <> template<class T> void A<int>::B<double>::mf1(T t) { }
    template <class Y> template <> template<class T> void A<Y>::B<double>::mf1(T t) { } // ill-formed; B<double> is specialized but
    template <class Y> template <> template<class T> void A<Y>::B<double>::mf1() { } // 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:

(20.1) — the explicit specialization of a function template;
(20.2) — the explicit specialization of a member function template;
(20.3) — 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 ]

17.9 Function template specializations [temp.fct.spec]

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
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

## 17.9.1 Explicit template argument specification

Template arguments can be specified when referring to a function template specialization 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>&)
}
```

```cpp
template<class U, class V> U convert(V v);
void g(double d) {
    int i = convert<int,double>(d); // int convert(double)
    char c = convert<char,double>(d); // char convert(double)
}
```

—end example

A template argument list may be specified when referring to a specialization of a function template

(2.1) — when a function is called,
(2.2) — 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,
(2.3) — in an explicit specialization,
(2.4) — in an explicit instantiation, or
(2.5) — in a friend declaration.

Trailing template arguments that can be deduced (17.9.2) or obtained from default template-arguments may be omitted from the list of explicit template-arguments. A trailing template parameter pack (17.6.3) not otherwise deduced will be deduced to an empty sequence of template arguments. If all of the template arguments can be deduced, they may all be omitted: in this case, the empty template argument list `<>` itself may also be omitted. In contexts where deduction is done and fails, or in contexts where deduction is not done, if a template argument list is specified and it, along with any default template arguments, identifies a single function template specialization, then the template-id is an lvalue for the function template specialization.

**Example:**

```cpp
template<class X, class Y> X f(Y);
template<class X, class Y, class ... Z> X g(Y);
void h() {
    int i = f<int>(5.6);         // Y is deduced to be double
    int j = f(5.6);              // ill-formed: X cannot be deduced
    f<void>(f<int,bool>);        // Y for outer f deduced to be 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 is deduced to be double, Z is deduced to an empty sequence
}
f<void>(g<int, bool>); // Y for outer f is deduced to be int (*)(bool), // Z is deduced to an empty sequence
}

—end example]

4 [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 (11.3.5) is visible that would otherwise be used. For example:

    template <class T> int f(T);  // #1
    int f(int);                    // #2
    int k = f(1);                   // uses #2
    int l = f<>(1);                 // uses #1

—end note]

5 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 is deduced to be double
        f<int,const char*>("aa",3.0);         // Y is deduced to be const char*, and Z is deduced to be double
        f("aa",3.0);                          // error: X cannot be deduced
        f2<char, short, int, long>();         // OK
    }

—end example]

6 Implicit conversions (Clause 7) 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]

7 [Note: Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. —end note]

8 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:

    temp.deduct]

    template<class ... Types> void f(Types ... values);
    void g() {
        f<int*, float*>(0, 0, 0);  // Types is deduced to the sequence int*, float*, int
    }

—end example]

17.9.2 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
template<class T> void f(T t); template<class X> void g(const X x); template<class Z> void h(Z, Z*);
```

```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

2 When an explicit template argument list is specified, if the template arguments are not compatible with the
template parameter list or do not result in a valid function type as described below, type deduction fails.
Specifically, the following steps are performed when evaluating an explicitly specified template argument list
with respect to a given function template:

(2.1) — If the specified template arguments do not match the template parameters in kind (i.e., type, non-type,
template), or if there are more arguments than there are parameters and no parameter is a template
parameter pack, or if there is not an argument for each non-pack parameter, type deduction fails.

(2.2) — If any non-type argument does not match the type of the corresponding non-type template parameter,
and is not convertible to the type of the corresponding non-type parameter as specified in 17.3.2, type
deduction fails.

(2.3) — The specified template argument values are substituted for the corresponding template parameters as
specified below.

3 After this substitution is performed, the function parameter type adjustments described in 11.3.5 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]

```cpp
template <class T> void f(T t);
```

```cpp
template <class Z> void h(Z, Z*);
```

```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

4 [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]

5 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:
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
    f(1, 'c');  // f<int,char>(1,'c')
    f(1);       // f<int,double>(1,0)
    f();        // error: T cannot be deduced
    f<int>();   // f<int,double>(0,0)
    f<int,char>(); // f<int,char>(0,0)
}

— end example

When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in the template parameter list of the template and the function type are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails. If the function template has associated constraints (17.4.2), those constraints are checked for satisfaction (17.4.1). 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. [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:

    template <class T> struct A { using X = typename T::X; };  
    template <class T> typename T::X f(typename A<T>::X);  
    template <class T> void f(...) { }  
    template <class T> auto g(typename A<T>::X) -> typename T::X;  
    template <class T> void g(...) { }  

    void h() {  
        f<int>(0);  // OK, substituting return type causes deduction to fail  
        g<int>(0);  // error, substituting parameter type instantiates A<int>  
    }  

— 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 and its template parameter types 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:

    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(...);
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:

— Attempting to instantiate a pack expansion containing multiple parameter packs of differing lengths.

— Attempting to create an array with an element type that is void, a function type, a reference type, or an abstract class 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]

— 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]

— Attempting to use a type in a nested-name-specifier of a qualified-id when that type does not contain the specified member, or

— the specified member is not a type where a type is required, or

— the specified member is not a template where a template is required, or

— the specified member is not a non-type where a non-type is required.

[Example:

  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<Y::N*>){}
  template <class T> void h(Z<T::template TT>){}
struct A {};
struct B { int Y; }
struct C {
    typedef int N;
};
struct D {
    typedef int TT;
};

int main() {
    // Deduction fails in each of these cases:
    f<A>(0);  // A does not contain a member Y
    f<B>(0);  // The Y member of B is not a type
    g<C>(0);  // The N member of C is not a non-type
    h<D>(0);  // The TT member of D is not a template
}

— end example ]

— Attempting to create a pointer to reference type.

— Attempting to create a reference to void.

— Attempting to create “pointer to member of T” when T is not a class type. [Example:
    template <class T> int f(int T::*);
    int i = f<int>(0);
— end example]

— Attempting to give an invalid type to a non-type template parameter. [Example:
    template <class T, T*> struct S {};
    template <class T> int f(S<T, T*>(*);
    struct X {};
    int i0 = f<X>(0);
— end example]

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [Example:
    template <class T, T*> int f(int);
    int i2 = f<int,1>(0);  // can’t conv 1 to int*
— end example]

— Attempting to create a function type in which a parameter has a type of void, or in which the return type is a function type or array type.

— Attempting to create a function type in which a parameter type or the return type is an abstract class type (13.4).

— end note]

12 [Example: In the following example, assuming a signed char cannot represent the value 1000, a narrowing conversion (11.6.4) would be required to convert the template-argument of type int to signed char, therefore substitution fails for the second template (17.3.2).

    template <int> int f(int);
    template <signed char> int f(int);
    int i1 = f<1000>(0);  // OK
    int i2 = f<i1>(0);  // ambiguous; not narrowing
— end example]
as a function template parameter type and the initializer element as its argument, and in the $P'[N]$ case, if $N$ is a non-type template parameter, $N$ is deduced from the length of the initializer list. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (17.9.2.5). [Example:

```cpp
template<class T> void f(std::initializer_list<T>){
  f({1,2,3}); // T deduced to int
  f({1,"asdf"}); // error: T deduced to both int and const char*
}
```

```cpp
template<class T> void g(T);
g({1,2,3}); // error: no argument deduced for T
```

```cpp
template<class T, int N> void h(T const&)[N];
h({1,2,3}); // T deduced to int, N deduced to 3
```

```cpp
template<class T> void j(T const&)[3];
j({42}); // error: no argument deduced for T
```

```cpp
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({1,2,3,4}); // OK, N deduced to 3
```

```cpp
template<int M, int N> void m(int const&)[M][N];
m({1,2,3,4}); // M and N both deduced to 2
```

```cpp
template<class T, int N> void n(T const&)[N], T);
n({1,2,3,4}, 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 (17.9.2.5), the type of that parameter pack is never deduced. [Example:

```cpp
template<... Types> void f(Types& ...);
template<class T1, ... Types> void g(T1, Types ...);
template<class T1, ... Types> void g1(Types ..., T1);
```

```cpp
void h(int x, float& y) {
  const int z = x;
  f(x, y, z); // Types is deduced to int, float, const int
  g(x, y, z); // T1 is deduced to int; Types is deduced to float, int
  g1(x, y, z); // error: Types is not deduced
  g1<int, int, int>(x, y, z); // OK, no deduction occurs
}
```

—end example]

2 If $P$ is not a reference type:

1. If $A$ is an array type, the pointer type produced by the array-to-pointer standard conversion (7.2) is used in place of $A$ for type deduction; otherwise,
2. If $A$ is a function type, the pointer type produced by the function-to-pointer standard conversion (7.3) is used in place of $A$ for type deduction; otherwise,
3. If $A$ is a cv-qualified type, the top-level cv-qualifiers of $A$'s type are ignored for type deduction.

3 If $P$ is a cv-qualified type, the top-level cv-qualifiers of $P$'s type are ignored for type deduction. If $P$ is a reference type, the type referred to by $P$ is used for type deduction. [Example:

```cpp
template<class T> int f(const T&);
int n1 = f(5); // calls f<int>(const int&)
const int i = 0;
int n2 = f(i); // calls f<int>(const int&)
template <class T> int g(volatile T&);
int n3 = g(i); // calls g<const int>(const volatile int&)
```
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 (16.3.1.8)). 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

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.
```

— end example]

In general, the deduction process attempts to find template argument values that will make the deduced A identical to A (after the type A is transformed as described above). However, there are three cases that allow a difference:

4.1 — If the original P is a reference type, the deduced A (i.e., the type referred to by the reference) can be more cv-qualified than the transformed A.

4.2 — The transformed A can be another pointer or pointer-to-member type that can be converted to the deduced A via a function pointer conversion (7.13) and/or qualification conversion (7.5).

4.3 — If P is a class and P has the form simple-template-id, then the transformed A can be a derived class 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 pointed to by the deduced A.

These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails. [Note: If a template-parameter is not used in any of the function parameters of a function template, or is used only in a non-deduced context, its corresponding template-argument cannot be deduced from a function call and the template-argument must be explicitly specified. — end note]  

5. When P is a function type, function-pointer type, or pointer-to-member-function type:

6.1 — If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.

6.2 — If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for the deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a non-deduced context.

7. [Example:

```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);  
    g(char);  
    int i = f(g); // calls f(int (*)(int))
```

— end example]

8. [Example:
Ambiguous deduction causes the second function parameter to be a non-deduced context.

```cpp
// Example:
// 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);
int i = f(1, g);  // calls f(int, int (*)(int))
```

If deduction succeeds for all parameters that contain `template-parameters` that participate in template argument deduction, and all template arguments are explicitly specified, deduced, or obtained from default template arguments, remaining parameters are then compared with the corresponding arguments. For each remaining parameter `P` with a type that was non-dependent before substitution of any explicitly-specified template arguments, if the corresponding argument `A` cannot be implicitly converted to `P`, deduction fails.

```cpp
// Example:
// 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))
```

17.9.2.2 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 (16.4). 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 17.9.2.5.

A placeholder type (10.1.7.4) 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.

17.9.2.3 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 11.6, 16.3.1.5, and 16.3.1.6 for the determination of that type) as described in 17.9.2.5.

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 (7.2) 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) 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.

When the deduction process requires a qualification conversion for a pointer or pointer-to-member type as described above, the following process is used to determine the deduced template argument values:

If \( A \) is a type
\[
\text{cv}_{0,0} \text{ “pointer to ...” } \text{cv}_{1,n-1} \text{ “pointer to” } \text{cv}_{1,n} \ T_1
\]
and \( P \) is a type
\[
\text{cv}_{2,0} \text{ “pointer to ...” } \text{cv}_{2,n-1} \text{ “pointer to” } \text{cv}_{2,n} \ T_2,
\]
then the cv-unqualified \( T_1 \) and \( T_2 \) are used as the types of \( A \) and \( P \) respectively for type deduction.

**Example:**
```cpp
struct A {
    template <class T> operator T***();
};
A a;
const int * const * const * p1 = a; // T is deduced as int, not const int
```

### 17.9.2.4 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 17.6.6.2. — 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.\(^{144}\)
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 (17.6.6.2) 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 \( A \). If a particular \( P \) contains no template-parameters that participate in template argument deduction, that \( P \) is not used to determine the ordering.

Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

1. If \( P \) is a reference type, \( P \) is replaced by the type referred to.

\(^{144}\) Default arguments are not considered to be arguments in this context; they only become arguments after a function has been selected.
If \(A\) is a reference type, \(A\) is replaced by the type referred to.

If both \(P\) and \(A\) were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

Remove any top-level cv-qualifiers:

If \(P\) is a cv-qualified type, \(P\) is replaced by the cv-unqualified version of \(P\).

If \(A\) is a cv-qualified type, \(A\) is replaced by the cv-unqualified version of \(A\).

Using the resulting types \(P\) and \(A\), the deduction is then done as described in 17.9.2.5. 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:

\[
\begin{align*}
\text{template<class... Args>} & \quad \text{void f(Args... args);} & \quad \text{// #1} \\
\text{template<class T1, class... Args>} & \quad \text{void f(T1 a1, Args... args);} & \quad \text{// #2} \\
\text{template<class T1, class T2>} & \quad \text{void f(T1 a1, T2 a2);} & \quad \text{// #3}\end{align*}
\]

\[f(); \quad \text{// calls #1}
\]

\[f(1, 2, 3); \quad \text{// calls #2}
\]

\[f(1, 2); \quad \text{// calls #3; non-variadic template #3 is more specialized}
\]

\[\text{than the variadic templates #1 and #2}\]

If, for a given type, deduction succeeds in both directions (i.e., 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):

- if the type from the argument template was an lvalue reference and the type from the parameter template was not, the parameter type is not considered to be at least as specialized as the argument type; otherwise,

- if the type from the argument template is more cv-qualified than the type from the parameter template (as described above), the parameter type is not considered to be at least as specialized as the argument type.

Function template \(F\) is at least as specialized as function template \(G\) if, for each pair of types used to determine the ordering, the type from \(F\) is at least as specialized as the type from \(G\). \(F\) is more specialized than \(G\) if \(F\) is at least as specialized as \(G\) and \(G\) is not at least as specialized as \(F\).

If, after considering the above, function template \(F\) is at least as specialized as function template \(G\) and vice-versa, and if \(G\) has a trailing parameter pack for which \(F\) does not have a corresponding parameter, and if \(F\) does not have a trailing 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.

\[\text{Note: A template parameter used in a non-deduced context is considered used.} \quad \text{— end note}\]

Example:

\[
\begin{align*}
\text{template <class T> T f(int);} & \quad \text{// #1} \\
\text{template <class T, class U> T f(U);} & \quad \text{// #2} \\
\text{void g()} \{
\text{f<int>(1);} & \quad \text{// calls #1}
\}
\end{align*}
\]

- 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:

\[
\begin{align*}
\text{template<class ...> struct Tuple { };} \\
\text{template<class ... Types> void g(Tuple<Types ...>);} & \quad \text{// #1} \\
\text{template<class T1, class ... Types> void g(Tuple<T1, Types ...>);} & \quad \text{// #2} \\
\text{template<class T1, class ... Types> void g(Tuple<T1, Types& ...>);} & \quad \text{// #3}\end{align*}
\]

§ 17.9.2.4
17.9.2.5 Deducing template arguments from a type

Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it $P$) is compared with an actual type (call it $A$), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make $P$, after substitution of the deduced values (call it the deduced $A$), compatible with $A$.

In some cases, the deduction is done using a single set of types $P$ and $A$, in other cases, there will be a set of corresponding types $P$ and $A$. Type deduction is done independently for each $P/A$ pair, and the deduced template argument values are then combined. If type deduction cannot be done for any $P/A$ pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails. The type of a type parameter is only deduced from an array bound if it is not otherwise deduced.

A given type $P$ can be composed from a number of other types, templates, and non-type values:

1. A function type includes the types of each of the function parameters and the return type.
2. A pointer-to-member type includes the type of the class object pointed to and the type of the member pointed to.
3. A type that is a specialization of a class template (e.g., $A<\text{int}>$) includes the types, templates, and non-type values referenced by the template argument list of the specialization.
4. An array type includes the array element type and the value of the array bound.

In most cases, the types, templates, and non-type values that are used to compose $P$ participate in template argument deduction. That is, they may be used to determine the value of a template argument, and template argument deduction fails if the value so determined is not consistent with the values determined elsewhere. In certain contexts, however, the value does not participate in type deduction, but instead uses the values of template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails. [Note: Under 17.9.2.1 and 17.9.2.4, if $P$ contains no template-parameters that appear in deduced contexts, no deduction is done, so $P$ and $A$ need not have the same form. —end note]

The non-deduced contexts are:

1. The nested-name-specifier of a type that was specified using a qualified-id.
2. The expression of a decltype-specifier.
3. A non-type template argument or an array bound in which a subexpression references a template parameter.
4. A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.
5. A function parameter for which argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (16.4), and one or more of the following apply:
   - more than one function matches the function parameter type (resulting in an ambiguous deduction), or
   - no function matches the function parameter type, or
   - the 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 (11.6.4) but the parameter does not have a type for which deduction from an initializer list is specified (17.9.2.1). [Example:

   ```cpp
template<class T> void g(T);
g({1,2,3}); // error: no argument deduced for T
``` - end example]
— A function parameter pack that does not occur at the end of the parameter-declaration-list.

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 \(\texttt{A<T>::B<T2>}\), both \(\texttt{T}\) and \(\texttt{T2}\) are non-deduced. Likewise, if a type is specified as \(\texttt{A\langle I+J\rangle::X<T>}\), \(\texttt{I}\), \(\texttt{J}\), and \(\texttt{T}\) are non-deduced. If a type is specified as \(\texttt{void f\langle\texttt{typename A<T>::B, A<T}\rangle}\), the \(\texttt{T}\) in \(\texttt{A<T>::B}\) is non-deduced but the \(\texttt{T}\) in \(\texttt{A<T>}\) 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 { /* ... */ }; // error: T could be A or B
struct B : A { /* ... */ }; // error: T could be & or B
void g(A a, B b) {
    f(a, b); // OK: T is A
    f(b, a); // OK: T is 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
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 { }; // calls f(B<int>&)
template <class T> struct D : public B<T> {}; // calls f(B<int>&)
struct D2 : public B<int> {}; // calls f(B<int>&)

void t() {
    D<int> d;
    D2 d2;
    f(d); // calls f(B<int>&)
    f(d2); // calls f(B<int>&)
}
```

— end example]

8 A template type argument \(T\), a template template argument \(TT\) or a template non-type argument \(i\) can be deduced if \(P\) and \(A\) have one of the following forms:

- \(T\)
- \(cv T\)
- \(T\star\)
- \(T&\)
- \(T&&\)
- \(T[\text{integer-constant}]\)
template-name<T> (where template-name refers to a class template)

T()
T(T)

T type::*
T::*
T T::*
T::*()()
type(T::*())()
type(T::*)(T)
T (T::*())
T (T::*)(T)
type[i]

where (T) represents a parameter-type-list (11.3.5) 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.

9 If P has a form that contains <T> or <i>, then each argument P_i of the respective template argument list of P is compared with the corresponding argument A_i of the corresponding template argument list of A. If the template argument list of P contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If P_i is a pack expansion, then the pattern of P_i is compared with each remaining argument in the template argument list of A_i. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by P_i. During partial ordering (17.9.2.4), if A_i was originally a pack expansion:

(9.1) — if P does not contain a template argument corresponding to A_i then A_i is ignored;
(9.2) — otherwise, if P_i is not a pack expansion, template argument deduction fails.

[Example:

```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, const T2&> { };     // #3
    S<int, const int&> s;     // both #2 and #3 match; #3 is more specialized
```

```
    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]

10 Similarly, if P has a form that contains (T), then each parameter type P_i of the respective parameter-type-list (11.3.5) 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 (17.9.2.2) or when deducing template arguments from a function declaration (17.9.2.6) and P_i and A_i are parameters of the top-level parameter-type-list of P and A, respectively, P_i is adjusted if it is a forwarding reference (17.9.2.1) and A_i is an lvalue reference, in which case the type of P_i is changed to be the template parameter type (i.e., T&& is changed to simply T). [Note: As a result, when P_i is T&& and A_i is X&, the adjusted P_i will be T, causing T to be deduced as X&. —end note] [Example:

```cpp
    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&k), i.e., #2
}

— end example]

If the parameter-declaration corresponding to Pi 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 (17.9.2.4), if Ai was originally a function parameter pack:

(10.1) — if P does not contain a function parameter type corresponding to Ai then Ai is ignored;
(10.2) — otherwise, if Pi 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 ]

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 ]

Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

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<T<n>> {
    using Q = T;
  };
  using R = long;
  using R = C<A<2>::Q;        // OK; T was deduced to 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 to std::size_t from the type int[42]
— end example ]

[ Example:
  template<class T, T i> void f(int (&a)[i]);
  int v[10];
  void g() {
    f(v);
    // OK: T is std::size_t
  }
— end example ]

§ 17.9.2.5 381
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]);
```

```cpp
void g() {
    int v[10][20];
    f1(v);       // OK: i deduced to be 20
    f1<20>(v);   // OK
    f2(v);       // error: cannot deduce template-argument i
    f2<10>(v);   // OK
    f3(v);       // OK: i deduced to be 10
}
```

—end note—

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>);
template <int i> void f(A<i>, A<i+1>);
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);  // 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.¹⁴⁵ [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
    f<1>(a);    // OK
}
```

¹⁴⁵ 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.

§ 17.9.2.5
template-argument can be deduced from a function, pointer to function, or pointer-to-member-function type.

Example:
```cpp
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
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
}
```

A template type-parameter cannot be deduced from the type of a function default argument. [Example:
```cpp
template <class T> void f(T = 5, T = 7);
void g() {
    f(1);  // OK: call f<int>(1,7)
    f();   // error: cannot deduce T
    f<int>();  // OK: call f<int>(5,7)
}
```

The template-argument corresponding to a template template-parameter is deduced from the type of the template-argument of a class template specialization used in the argument list of a function call. [Example:
```cpp
template <template <class T> class X> struct A { };  
template <template <class T> class X> void f(A<X>) { }  
template<class T> struct B { };  
A<B> ab;  
f(ab);  // calls f(A<B>)
```

[Note: Template argument deduction involving parameter packs (17.6.3) can deduce zero or more arguments for each parameter pack. — end note] [Example:
```cpp
template<class> struct X { };  
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { };  
template<class ... Types> struct Y { };  
template<class T, class ... Types> struct Y<T, Types ...> { };  

template<class ... Types> int f(void (*)(Types ...));
void g(int, float);
```
```cpp
X<int> x1;  // uses primary template
X<int(int, float, double)> x2;  // uses partial specialization; ArgTypes contains float, double
X<int(float, int)> x3;  // uses primary template
Y<int> y1;  // use primary template; Types is empty
Y<int, float&, double&> y2;  // uses partial specialization; T is int&, Types contains float, double
Y<int, float, double> y3;  // uses primary template; Types contains int, float, double
int fv = f(g);  // OK; Types contains int, float
```

—end example]
17.9.2.6 Deducing template arguments from a function declaration

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 (17.8.2), explicit specializations (17.8.3), and certain friend declarations (17.6.4). This is also done to determine whether a deallocation function template specialization matches a placement operator new (6.6.4.1.2, 8.5.2.4). 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 8.5.2.4. The deduction is done as described in 17.9.2.5.

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 (17.6.6.2), deduction fails and, in the declaration cases, the program is ill-formed.

17.9.3 Overload resolution

A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (17.9.2) and checking of any explicit template arguments (17.3) 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 16.3.3.146

[Example:

template<class T> T max(T a, T b) { return a>b?a:b; }

  void f(int a, int b, char c, char d) {
    int m1 = max(a,b); // max(int, int)
    char m2 = max(c,d); // max(char, char)
    int m3 = max(a,c);  // error: cannot generate max(int, char)
  }

Adding the non-template function

    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:

    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)
    }

    — end example]

146) 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 16.3.3 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 `f` is well-formed even if the template `f` is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for `f<const char*>`, either implicitly or explicitly generated, is present in some translation unit. —end example]

17.10 Deduction guides

deduction-guide:
    explicit_opt template-name ( parameter-declaration-clause ) -> simple-template-id ;

1 Deduction guides are used when a `template-name` appears as a type specifier for a deduced class type (10.1.7.5). Deduction guides are not found by name lookup. Instead, when performing class template argument deduction (16.3.1.8), any deduction guides declared for the class template are considered.

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 `parameter-declaration-clause` of a deduction guide as in a function declaration (11.3.5). The `simple-template-id` shall name a class template specialization. The `template-name` shall be the same identifier as the `template-name` of the `simple-template-id`. A `deduction-guide` shall be declared in the same scope as the corresponding class template and, for a member class template, with the same access. Two deduction guide declarations in the same translation unit for the same class template shall not have equivalent `parameter-declaration-clauses`. 

§ 17.10
18 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 (18.3).

A **try-block** is a statement (Clause 9). [Note: Within this Clause “try block” is taken to mean both try-block
and function-try-block. —end note]

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]

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]

In this Clause, “before” and “after” refer to the “sequenced before” relation (6.8.1).

18.1 Throwing an exception [except.throw]

1 Throwing an exception transfers control to a handler. [Note: An exception can be thrown from one of the following contexts: throw-expressions (8.5.17), allocation functions (6.6.4.4.1), dynamic_cast (8.5.1.7), typeid (8.5.1.8), new-expressions (8.5.2.4), and standard library functions (20.4.1.4). — 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 (18.3); “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 (11.6, 15.8) a temporary object, called the exception object. An lvalue denoting the temporary is used to initialize the variable declared in the matching handler (18.3). If the type of the exception object would be an incomplete type or a pointer to an incomplete type other than cv void the program is ill-formed.

§ 18.1
The memory for the exception object is allocated in an unspecified way, except as noted in 6.6.4.4.1. If a
handler exits by rethrowing, control is passed to another handler for the same exception object. The points
of potential destruction for the exception object are:

(4.1) when an active handler for the exception exits by any means other than rethrowing, immediately after
the destruction of the object (if any) declared in the exception-declaration in the handler;

(4.2) when an object of type std::exception_ptr (21.8.6) 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 21.8.6 and 33.6. — 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 (15.8). The destructor is potentially invoked (15.4).

An exception is considered caught when a handler for that exception becomes active (18.3). [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 (18.5.2) directly invokes a function
that exits via an exception, std::terminate is called (18.5.1). [Example:

```
struct C {
    C() { }
    C(const C&) {
        if (std::uncaught_exceptions()) {
            throw 0;  // throw during copy to handler's exception-declaration object (18.3)
        }
    }
};

int main() {
    try {
        throw C();  // calls std::terminate() if construction of the handler's
                    // exception-declaration object is not elided (15.8)
    } catch(C) { }

    — end example] [Note: Consequently, destructors should generally catch exceptions and not let them
    propagate. — end note]

18.2 Constructors and destructors [except.ctor]

As control passes from the point where an exception is thrown to a handler, destructors are invoked by a
process, specified in this subclause, called stack unwinding.

The destructor is invoked for each automatic object of class type 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 (9.6.3), 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:

```
struct A { };

struct Y { ~Y() noexcept(false) { throw 0; } };

A f() {
    try {
        A a;
        Y y;
        A b;
        return {};
    } catch (...) { 
}
```

§ 18.2 388
At #1, the returned object of type A is constructed. Then, the local variable b is destroyed (9.6). 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.

---

3 If the initialization or destruction of an object other than by delegating constructor is terminated by an exception, the destructor is invoked for each of the object’s direct subobjects and, for a complete object, virtual base class subobjects, whose initialization has completed (11.6) 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. The subobjects are destroyed in the reverse order of the completion of their construction. Such destruction is sequenced before entering a handler of the function-try-block of the constructor or destructor, if any.

4 If the compound-statement of the function-body of a delegating constructor for an object exits via an exception, the object’s destructor is invoked. Such destruction is sequenced before entering a handler of the function-try-block of a delegating constructor for that object, if any.

[Note: If the object was allocated by a new-expression (8.5.2.4), the matching deallocation function (6.6.4.4.2), if any, is called to free the storage occupied by the object. — end note]

18.3 Handling an exception

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

- 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
- the handler is of type cv T or cv T& and T is an unambiguous public base class of E, or
- the handler is of type cv T or const T where T is a pointer or pointer-to-member type that can be converted to T by one or more of
  - a standard pointer conversion (7.11) not involving conversions to pointers to private or protected or ambiguous classes
  - a function pointer conversion (7.13)
  - a qualification conversion (7.5), or
- 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 (8.5.17). — end note]

[Example:

class Matherr { /* ... */ virtual void vf(); }; 
class Overflow: public Matherr { /* ... */ }; 
class Underflow: public Matherr { /* ... */ }; 
class Zerodivide: public Matherr { /* ... */ };

doctor f() {
    try {
        g();
    } catch (Overflow oo) {
        // ...
    } catch (Matherr mm) {
        // ...
    }
}
Here, the Overflow handler will catch exceptions of type Overflow and the Matherr handler will catch exceptions of type Matherr and of all types publicly derived from Matherr including exceptions of type Underflow and Zerodivide. — end example]

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 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 (18.5.1).

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 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 9.6.3).

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 (11.6) from the corresponding base class subobject of the exception object;

(15.2) — otherwise, the variable is copy-initialized (11.6) 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.

18.4 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 (11.3.5).

```cpp
noexcept-specifier:
  noexcept ( constant-expression )
  noexcept
  throw ( )
```
In a **noexcept-specifier**, the **constant-expression**, if supplied, shall be a contextually converted constant expression of type `bool` (8.6); 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 (11.6). The **noexcept-specifier** **noexcept** without a **constant-expression** is equivalent to the **noexcept-specifier** **noexcept**(true). The **noexcept-specifier** **throw**() is deprecated (D.3), and equivalent to the **noexcept-specifier** **noexcept**(true).

If a declaration of a function does not have a **noexcept-specifier**, the declaration has a potentially throwing exception specification unless it is a destructor or a deallocation function or is defaulted on its first declaration, in which cases the exception specification is as specified below and no other declaration for that function shall have a **noexcept-specifier**. In an explicit instantiation (17.8.2) 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.

If a virtual function has a non-throwing exception specification, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall have a non-throwing exception specification, unless the overriding function is defined as deleted. [Example:

```cpp
struct B {
    virtual void f() noexcept;
    virtual void g();
    virtual void h() noexcept = delete;
};

struct D: B {
    void f();          // 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]

Whenever an exception is thrown and the search for a handler (18.3) encounters the outermost block of a function with a non-throwing exception specification, the function `std::terminate()` is called (18.5.1). [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:

```cpp
extern void f();           // potentially-throwing

void g() noexcept {
    f();         // valid, even if f throws
    throw 42;    // valid, effectively a call to std::terminate
}
```

The call to f is well-formed even though, when called, f might throw an exception. — end example]

An expression e is potentially-throwing if

(6.1) — e is a function call (8.5.1.2) whose **postfix-expression** has a function type, or a pointer-to-function type, with a potentially-throwing exception specification, or

(6.2) — e implicitly invokes a function (such as an overloaded operator, an allocation function in a **new-expression**, a constructor for a function argument, or a destructor if e is a full-expression (6.8.1)) that is potentially-throwing, or

(6.3) — e is a **throw-expression** (8.5.17), or

(6.4) — e is a **dynamic_cast** expression that casts to a reference type and requires a runtime check (8.5.1.7), or

(6.5) — e is a **typeid** expression applied to a (possibly parenthesized) built-in unary * operator applied to a pointer to a polymorphic class type (8.5.1.8), or

(6.6) — 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 (18.2), their exception specifications do not contribute to the exception specification of the constructor, because an exception thrown from such a destructor would call std::terminate rather than escape the constructor (18.1, 18.5.1). — end note]

8 The exception specification for an implicitly-declared destructor, or a destructor without a noexcept-specifier, is potentially-throwing if and only if any of the destructors for any of its potentially constructed subobjects is potentially throwing.

9 The exception specification for an implicitly-declared assignment operator, or an assignment-operator without a noexcept-specifier that is defaulted on its first declaration, is potentially-throwing if and only if the invocation of any assignment operator in the implicit definition is potentially-throwing.

10 A deallocation function (6.6.4.4.2) with no explicit noexcept-specifier has a non-throwing exception specification.

11 The exception specification for a comparison operator (8.5.8, 8.5.9, 8.5.10) 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.

12 [Example:

```cpp
struct A {
    A(int = (A(5), 0)) noexcept;
    A(const A&) noexcept;
    A(A&&) noexcept;
    ~A();
};
struct B {
    B() throw();
    B(const B&) = default; // implicit exception specification is noexcept(true)
    B(B&&, int = (throw Y(), 0)) noexcept;
    ~B() noexcept(false);
};
```

```cpp
struct D : public A, public B {
    int * p = new int[n];
    // D::D() potentially-throwing, as the new operator may throw bad_alloc or bad_array_new_length
    // D::D(const D&) non-throwing
    // D::D(D&&) potentially-throwing, as the default argument for B's constructor may throw
    // D::~D() 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]

13 An exception specification is considered to be needed when:

(13.1) — in an expression, the function is the unique lookup result or the selected member of a set of overloaded functions (6.4, 16.3, 16.4);

(13.2) — 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;

(13.3) — the exception specification is compared to that of another declaration (e.g., an explicit specialization or an overriding virtual function);

(13.4) — the function is defined; or

(13.5) — 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.

18.5 Special functions [except.special]

1 The function std::terminate() (18.5.1) is used by the exception handling mechanism for coping with errors related to the exception handling mechanism itself. The function std::current_exception() (21.8.6) and the class std::nested_exception (21.8.7) can be used by a program to capture the currently handled exception.

18.5.1 The std::terminate() function [except.terminate]

1 In some situations exception handling must be abandoned for less subtle error handling techniques. [Note: These situations are:

(1.1) — when the exception handling mechanism, after completing the initialization of the exception object but before activation of a handler for the exception (18.1), calls a function that exits via an exception, or

(1.2) — when the exception handling mechanism cannot find a handler for a thrown exception (18.3), or

(1.3) — when the search for a handler (18.3) encounters the outermost block of a function with a non-throwing exception specification (18.4), or

(1.4) — when the destruction of an object during stack unwinding (18.2) terminates by throwing an exception, or

(1.5) — when initialization of a non-local variable with static or thread storage duration (6.8.3.3) exits via an exception, or

(1.6) — when destruction of an object with static or thread storage duration exits via an exception (6.8.3.4), or

(1.7) — when execution of a function registered with std::atexit or std::at_quick_exit exits via an exception (21.5), or

(1.8) — when a throw-expression (8.5.17) with no operand attempts to rethrow an exception and no exception is being handled (18.1), or

(1.9) — when the function std::nested_exception::rethrow_nested is called for an object that has captured no exception (21.8.7), or

(1.10) — when execution of the initial function of a thread exits via an exception (33.3.2.2), or

(1.11) — for a parallel algorithm whose ExecutionPolicy specifies such behavior (23.19.4, 23.19.5, 23.19.6), when execution of an element access function (28.4.1) of the parallel algorithm exits via an exception (28.4.4), or

(1.12) — when the destructor or the copy assignment operator is invoked on an object of type std::thread that refers to a joinable thread (33.3.2.3, 33.3.2.4), or

(1.13) — when a call to a wait(), wait_until(), or wait_for() function on a condition variable (33.5.3, 33.5.4) fails to meet a postcondition.

—end note]

2 In such cases, std::terminate() is called (21.8.4). 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 (18.3) encounters the outermost block of a function with a non-throwing exception specification (18.4), it is implementation-defined whether the stack is unwound, unwound partially, or not unwound at all before std::terminate() is called. In all other situations, the stack shall not be unwound before 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 std::terminate().

18.5.2 The std::uncaught_exceptions() function [except.uncaught]

1 An exception is considered uncaught after completing the initialization of the exception object (18.1) until completing the activation of a handler for the exception (18.3). This includes stack unwinding. If an exception
is rethrown (8.5.17, 21.8.6), it is considered uncaught from the point of rethrow until the rethrown exception is caught. The function `std::uncaught_exceptions()` (21.8.5) returns the number of uncaught exceptions in the current thread.
19 Preprocessing directives

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) is either
the first character in the source file (optionally after white space containing no new-line characters) or that
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. A new-line character ends the preprocessing
directive even if it occurs within what would otherwise be an invocation of a function-like macro.

147) 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 19.3.2,
for example).
2 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.

3 When in a group that is skipped (19.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

4 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).

5 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called preprocessing, because conceptually they occur before translation of the resulting translation unit.

6 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]
The third and fourth forms of has-include-expression are considered only if neither of the first or second
forms matches, 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.

The #ifdef and #ifndef directives, and the defined conditional inclusion operator, shall treat __has_-include as if it were the name of a defined macro. The identifier __has_include 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

```
#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 8.6 using arithmetic that has at least the ranges specified in 21.3. 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 (21.4). [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 is not guaranteed to evaluate to the same value in these two contexts:

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

— end note] Also, whether a single-character character literal may have a negative value is implementation-
defined. Each subexpression with type bool is subjected to integral promotion before processing continues.

Preprocessing directives of the forms

```
#define identifier new-line groupopt
#undef identifier new-line groupopt
```

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent
to #if defined identifier and #if !defined identifier respectively.

Each directive’s condition is checked in order. If it evaluates to false (zero), the group that it controls is
skipped: directives are processed only through the name that determines the directive in order to keep track
of the level of nested conditionals; the rest of the directives’ preprocessing tokens are ignored, as are the other
preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero)
is processed; any following groups are skipped and their controlling directives are processed as if they were in
a group that is skipped. If none of the conditions evaluates to true, and there is a #else directive, the

§ 19.1
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 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.

[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"

149) 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.

150) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.2); thus, an expansion that results in two string literals is an invalid directive.
§ 19.3  Macro replacement  [cpp.replace]

1 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.

2 An identifier currently defined as an object-like macro (see below) may be redefined by another #define preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed. Likewise, an identifier currently defined as a function-like macro (see below) may be redefined by another #define preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.

3 There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be at least as many arguments in the invocation as there are parameters in the macro definition (excluding the ...). There shall exist a ) preprocessing token that terminates the invocation.

5 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.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

7 The identifier immediately following the define is called the 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.

8 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.

9 A preprocessing directive of the form
   
   # define identifier replacement-list new-line

   defines an object-like macro that causes each subsequent instance of the macro name to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

   # define identifier (paren identifier-listopt ) replacement-list new-line
   # define identifier (paren ... ) replacement-list new-line
   # define identifier (paren identifier-list , ... ) replacement-list new-line

   defines a function-like macro with parameters, whose use is similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the #define preprocessing directive. Each subsequent instance of the function-like macro name followed by a ( as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation

--- end note]
of the macro). The replaced sequence of preprocessing tokens is terminated by the matching ) preprocessing
token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the
sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a
normal white-space character.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of
arguments for the function-like macro. The individual arguments within the list are separated by comma
preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate
arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise
act as preprocessing directives, the behavior is undefined.

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 . . .).

19.3.1 Argument substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution
takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed
by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained
therein have been expanded. Before being substituted, each argument’s preprocessing tokens are completely
macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

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.

The identifier __VA_OPT__ shall always occur as part of the token sequence __VA_OPT__(content), where
content is an arbitrary sequence of preprocessing-tokens other than __VA_OPT__ which is terminated by the
closing ) and skips intervening pairs of matching left and right parentheses. If content would be ill-formed
as the replacement list of the current function-like macro, the program is ill-formed. The token sequence
__VA_OPT__(content) shall be treated as if it were a parameter, and the preprocessing tokens used to replace
it are defined as follows. If the variable arguments consist of no tokens, the replacement consists of a single
placemarker preprocessing token (19.3.3, 19.3.4). Otherwise, the replacement consists of the results of the
expansion of content as the replacement list of the current function-like macro before rescanning and further
replacement. [Example:

```c
#define F(...) f(0 __VA_OPT__(, __VA_ARGS__))
#define G(X, ...) f(0, X __VA_OPT__(,) __VA_ARGS__)
#define SDEF(sname, ...) S sname __VA_OPT__(= { __VA_ARGS__ })
F(a, b, c)    // replaced by f(0, a, b, c)
F()          // 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 (19.3.3)
#define H2(X, Y, ...) __VA_OPT__(X ## Y,) __VA_ARGS__
H2(a, b, c, d)    // replaced by ab, c, d

-- end example]
```

19.3.2 The # operator

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.

---

153) A conditionally-supported-directive is a preprocessing directive regardless of whether the implementation supports it.
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. Each occurrence of white space between the 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 argument is deleted. Otherwise, the original spelling of each preprocessing token in the 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 argument is `""`. The order of evaluation of `#` and `##` operators is unspecified.

19.3.3 The **##** operator

1 A `##` preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a `##` preprocessing token, the parameter is replaced by the corresponding argument’s preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.154

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a `##` preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially: concatenation of two 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:

```c
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the `##` operator. — end example]

19.3.4 Rescanning and further replacement

1 After all parameters in the replacement list have been substituted and `#` and `##` processing has taken place, all placemarker preprocessing tokens are removed. Then the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2 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.

3 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 19.9 below.

154) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.
19.3.5  Scope of macro definitions

1 A macro definition lasts (independent of block structure) until a corresponding \#undef directive is encountered or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after translation phase 4.

2 A preprocessing directive of the form

\# undef identifier new-line

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 [Example: The simplest use of this facility is to define a “manifest constant”, as in

```c
#define TABSIZE 100
int table[TABSIZE];
```

—end example]

4 [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.

```c
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly. —end example]

5 [Example: To illustrate the rules for redefinition and reexamination, the sequence

```c
#define x 3
#define f(a) f(x * (a))
#undef x
#define x 2
#define g f
#define z x[0]
#define h g(~
#define m(a) a(w)
#define v 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(x)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
(f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(5), r(,)};
char c[2][6] = { str(hello), str() };
results in
```

```c
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] = { 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

```c
#define str(s) # s
#define xstr(s) str(s)
#define debug(s, t) printf("x" # s "=" %d, x" # t "=" %s, x ## 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"
```

§ 19.3.5 402
Example: To illustrate the rules for placemarker preprocessing tokens, the sequence

```c
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
           t(10,,), t(,,11), t(,,12), t(,,) };
```
results in

```c
int j[] = { 123, 45, 67, 89,
           10, 11, 12, };
```

—end example

Example: To demonstrate the redefinition rules, the following sequence is valid.

```c
#define OBJ_LIKE (1-1)
#define OBJ_LIKE /* white space */ (1-1) /* other */
#define FUNC_LIKE(a) ( a )
#define FUNC_LIKE( a )( /* note the white space */
     a /* other stuff on this line */
   )
```
But the following redefinitions are invalid:

```c
#define OBJ_LIKE (0) // different token sequence
#define OBJ_LIKE (1 - 1) // different white space
#define FUNC_LIKE(b) ( a ) // different parameter usage
#define FUNC_LIKE(b) ( b ) // different parameter spelling
```
—end example

Example: Finally, to show the variable argument list macro facilities:

```c
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))
```

results in

```c
fprintf(stderr, "Flag");
printf(stderr, "X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```
—end example


19.4 Line control

The string literal of a `#line` directive, if present, shall be a character string literal.

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.

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.

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.

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.

19.5 Error directive

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.

19.6 Pragma directive

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.

19.7 Null directive

A preprocessing directive of the form

```
# new-line
```

has no effect.

19.8 Predefined macro names

The following macro names shall be defined by the implementation:

\[__cplusplus\]

The integer literal \[201703L\].

\[__DATE__\]

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.

\[__DATE__\]

It is intended that future versions of this International Standard will replace the value of this macro with a greater value. Non-conforming compilers should use a value with at most five decimal digits.
The presumed name of the current source file (a character string literal).\textsuperscript{156}

The presumed line number (within the current source file) of the current source line (an integer literal).\textsuperscript{157}

The integer literal \texttt{1} if the implementation is a hosted implementation or the integer literal \texttt{0} if it is not.

An integer literal of type \texttt{std::size_t} whose value is the alignment guaranteed by a call to \texttt{operator new(std::size_t)} or \texttt{operator new[](std::size_t)}. \texttt{[Note: Larger alignments will be passed to operator new(std::size_t, std::align_val_t), etc. (8.5.2.4). — end note]}

The time of translation of the source file: a character string literal of the form "\texttt{hh:mm:ss}" as in the time generated by the \texttt{asctime} function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

The following macro names are conditionally defined by the implementation:

Whether \texttt{__STDC__} is predefined and if so, what its value is, are implementation-defined.

The integer literal \texttt{1}, intended to indicate that, in the encoding for \texttt{wchar_t}, a member of the basic character set need not have a code value equal to its value when used as the lone character in an ordinary character literal.

Whether \texttt{__STDC_VERSION__} is predefined and if so, what its value is, are implementation-defined.

An integer literal of the form \texttt{yyyymmL} (for example, \texttt{199712L}). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type \texttt{wchar_t}, has the same value as the short identifier of that character. The \textit{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.

Defined, and has the value integer literal \texttt{1}, if and only if the implementation has strict pointer safety (6.6.4.4.3).

Defined, and has the value integer literal \texttt{1}, if and only if a program can have more than one thread of execution (6.8.2).

The values of the predefined macros (except for \texttt{__FILE__} and \texttt{__LINE__}) remain constant throughout the translation unit.

If any of the pre-defined macro names in this subclause, or the identifier \texttt{defined}, is the subject of a \texttt{#define} or a \texttt{undef} preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

\textsuperscript{156} The presumed source file name can be changed by the \texttt{#line} directive.

\textsuperscript{157} The presumed line number can be changed by the \texttt{#line} directive.
A unary operator expression of the form:

```
Pragma ( string-literal )
```

is processed as follows: The string literal is destringized by deleting the L prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the `pp-tokens` in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

Example:

```
#pragma listing on ..\listing.dir
```

can also be expressed as:

```
Pragma ( "listing on ..\listing.dir" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )
```

—end example]
20 Library introduction

20.1 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 (20.3), method of description (20.4), and organization (20.5.1) of the library. 20.5, Clause 21 through Clause 33, 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 21–Clause 33, as shown in Table 15.

Table 15 — Library categories

<table>
<thead>
<tr>
<th>Clause</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clause 21</td>
<td>Language support library</td>
</tr>
<tr>
<td>Clause 22</td>
<td>Diagnostics library</td>
</tr>
<tr>
<td>Clause 23</td>
<td>General utilities library</td>
</tr>
<tr>
<td>Clause 24</td>
<td>Strings library</td>
</tr>
<tr>
<td>Clause 25</td>
<td>Localization library</td>
</tr>
<tr>
<td>Clause 26</td>
<td>Containers library</td>
</tr>
<tr>
<td>Clause 27</td>
<td>Iterators library</td>
</tr>
<tr>
<td>Clause 28</td>
<td>Algorithms library</td>
</tr>
<tr>
<td>Clause 29</td>
<td>Numerics library</td>
</tr>
<tr>
<td>Clause 30</td>
<td>Input/output library</td>
</tr>
<tr>
<td>Clause 31</td>
<td>Regular expressions library</td>
</tr>
<tr>
<td>Clause 32</td>
<td>Atomic operations library</td>
</tr>
<tr>
<td>Clause 33</td>
<td>Thread support library</td>
</tr>
</tbody>
</table>

The language support library (Clause 21) provides components that are required by certain parts of the C++ language, such as memory allocation (8.5.2.4, 8.5.2.5) and exception processing (Clause 18).

The diagnostics library (Clause 22) provides a consistent framework for reporting errors in a C++ program, including predefined exception classes.

The general utilities library (Clause 23) includes components used by other library elements, such as a predefined storage allocator for dynamic storage management (6.6.4.4), and components used as infrastructure in C++ programs, such as tuples, function wrappers, and time facilities.

The strings library (Clause 24) provides support for manipulating text represented as sequences of type char, 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 25) provides extended internationalization support for text processing.

The containers (Clause 26), iterators (Clause 27), and algorithms (Clause 28) libraries provide a C++ program with access to a subset of the most widely used algorithms and data structures.

The numerics library (Clause 29) 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 30) 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 31) provides regular expression matching and searching.
The atomic operations library (Clause 32) allows more fine-grained concurrent access to shared data than is possible with locks.

The thread support library (Clause 33) provides components to create and manage threads, including mutual exclusion and interthread communication.

20.2 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.

20.3 Definitions

[Note: Clause 3 defines additional terms used elsewhere in this document. —end note]

20.3.1 arbitrary-positional stream

stream (described in Clause 30) that can seek to any integral position within the length of the stream

[Note 1 to entry: Every arbitrary-positional stream is also a repositional stream. —end note]

20.3.2 character

(Clause 24, Clause 25, Clause 30, and Clause 31) object which, when treated sequentially, can represent text

[Note 1 to entry: The term does not mean only `char`, `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]

20.3.3 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]

20.3.4 comparison function

operator function (16.5) for any of the equality (8.5.10) or relational (8.5.9) operators

20.3.5 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]

20.3.6 constant subexpression

expression whose evaluation as subexpression of a `conditional-expression` `CE` (8.5.16) would not prevent `CE` from being a core constant expression (8.6)

20.3.7 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

20.3.8 default behavior

(implementation) specific behavior provided by the implementation, within the scope of the `required behavior`
20.3.9 default behavior
<specification> description of replacement function and handler function semantics

20.3.10 direct-non-list-initialization
direct-initialization (11.6) that is not list-initialization (11.6.4)

20.3.11 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 21). — end note ]

20.3.12 iostream class templates
<templates, defined in Clause 30, that take two template arguments>
[ Note 1 to entry: The arguments are named charT and traits. The argument charT is a character container class, and the argument traits is a class which defines additional characteristics and functions of the character type represented by charT necessary to implement the iostream class templates. — end note ]

20.3.13 modifier function
class member function (12.2.1) other than a constructor, assignment operator, or destructor that alters the state of an object of the class

20.3.14 move assignment
assignment of an rvalue of some object type to a modifiable lvalue of the same type

20.3.15 move construction
direct-initialization of an object of some type with an rvalue of the same type

20.3.16 NTCTS
<sequence of values that have character type that precede the terminating null character type value charT()>

20.3.17 observer function
class member function (12.2.1) that accesses the state of an object of the class but does not alter that state
[ Note 1 to entry: Observer functions are specified as const member functions (12.2.2.1). — end note ]

20.3.18 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 ]

20.3.19 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 ]
20.3.20 repositional stream
stream (described in Clause 30) that can seek to a position that was previously encountered

20.3.21 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]

20.3.22 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]

20.3.23 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 20.5.5.7. — end note]

20.3.24 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

20.3.25 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]

20.4 Method of description (Informative)

This subclause describes the conventions used to specify the C++ standard library. 20.4.1 describes the structure of the normative Clause 21 through Clause 33 and Annex D, 20.4.2 describes other editorial conventions.

20.4.1 Structure of each clause

20.4.1.1 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

20.4.1.2 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.

2 The contents of the summary and the detailed specifications include:

(2.1) macros

158 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.
Requirements describe constraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

1. Template arguments
2. Derived classes
3. Containers, iterators, and algorithms that meet an interface convention

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 a table that specifies an initial set of the valid expressions and their semantics. Any generic algorithm (Clause 28) that uses the well-defined expression requirements is described in terms of the valid expressions for its template type parameters.

Template argument requirements are sometimes referenced by name. See 20.4.2.1.

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.\(^\text{159}\)

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):\(^\text{160}\)

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):\(^\text{161}\)

1. Requires: the preconditions for calling the function
2. Effects: the actions performed by the function
3. Synchronization: the synchronization operations (6.8.2) applicable to the function

---

\(^\text{159}\) Although in some cases the code given is unambiguously the optimum implementation.

\(^\text{160}\) 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.

\(^\text{161}\) To save space, items that do not apply to a function are omitted. For example, if a function does not specify any further preconditions, there will be no Requires: paragraph.
— **Postconditions:** the observable results established by the function

— **Returns:** a description of the value(s) returned by the function

— **Throws:** any exceptions thrown by the function, and the conditions that would cause the exception

— **Complexity:** the time and/or space complexity of the function

— **Remarks:** additional semantic constraints on the function

— **Error conditions:** the error conditions for error codes reported by the function

Whenever the Effects element specifies that the semantics of some function $F$ are *Equivalent to* some code sequence, then the various elements are interpreted as follows. If $F$’s semantics specifies a Requires element, then that requirement is logically imposed prior to the equivalent-to semantics. Next, the semantics of the code sequence are determined by the Requires, Effects, Synchronization, Postconditions, Returns, Throws, Complexity, Remarks, and Error conditions specified for the function invocations contained in the code sequence. The value returned from $F$ is specified by $F$’s Returns element, or if $F$ has no Returns element, a non-void return from $F$ is specified by the return statements in the code sequence. If $F$’s semantics contains a Throws, Postconditions, or Complexity element, then that supersedes any occurrences of that element in the code sequence.

For non-reserved replacement and handler functions, Clause 21 specifies two behaviors for the functions in question: their required and default behavior. The default behavior describes a function definition provided by the implementation. The required behavior describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior.

If the formulation of a complexity requirement calls for a negative number of operations, the actual requirement is zero operations.\(^{162}\)

Complexity requirements specified in the library clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.

Error conditions specify conditions where a function may fail. The conditions are listed, together with a suitable explanation, as the `enum class errc` constants (22.5).

20.4.1.5 C library [structure.see.also]

Paragraphs labeled “See also” contain cross-references to the relevant portions of the ISO C standard.

20.4.2 Other conventions [conventions]

This subclause describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (20.4.2.1), and member functions (20.4.2.2).

20.4.2.1 Type descriptions [type.descriptions]

20.4.2.1.1 General [type.descriptions.general]

The Requirements subclauses may describe names that are used to specify constraints on template arguments.\(^{163}\) 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 30 are used to describe implementation-defined types. They are based on other types, but with added constraints.

20.4.2.1.2 Exposition-only types [expos.only.types]

Several types defined in Clause 21 through Clause 33 and Annex D that are used as function parameter or return types are defined for the purpose of exposition only in order to capture their language linkage. The declarations of such types are followed by a comment ending in *exposition only*. [Example:

```cpp
namespace std {
    extern "C" using some-handler = int(int, void*, double); // exposition only
}
```

The type placeholder `some-handler` can now be used to specify a function that takes a callback parameter with C language linkage. — end example]

\(^{162}\) This simplifies the presentation of complexity requirements in some cases.

\(^{163}\) Examples from 20.5.3 include: `EqualityComparable`, `LessThanComparable`, `CopyConstructible`. Examples from 27.2 include: `InputIterator`, `ForwardIterator`.

§ 20.4.2.1.2
20.4.2.1.3 Enumerated types  

Several types defined in Clause 30 are *enumerated types*. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.\(^{164}\)

The enumerated type *enumerated* can be written:

```c
enum enumerated { V0, V1, V2, V3, ...; }
```

```c
inline const enumerated C0(V0);
inline const enumerated C1(V1);
inline const enumerated C2(V2);
inline const enumerated C3(V3);
......
```

Here, the names \(C_0, C_1, \ldots\) represent *enumerated elements* for this particular enumerated type. All such elements have distinct values.

20.4.2.1.4 Bitmask types  

Several types defined in Clause 21 through Clause 33 and Annex D are *bitmask types*. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a bitset (23.9.2).

The bitmask type *bitmask* can be written:

```c
// For exposition only.
// int_type is an integral type capable of representing all values of the bitmask type.
enum bitmask : int_type {
    V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, ...
};
```

```c
inline constexpr bitmask C0(V0);
inline constexpr bitmask C1(V1);
inline constexpr bitmask C2(V2);
inline constexpr bitmask C3(V3);
......
```

Here, the names \(C_0, C_1, \ldots\) represent *bitmask elements* for this particular bitmask type. All such elements have distinct, nonzero values such that, for any pair \(C_i\) and \(C_j\) where \(i \neq j\), \(C_i \& C_j\) is nonzero and \(C_i \& C_j\) is zero. Additionally, the value 0 is used to represent an *empty bitmask*, in which no bitmask elements are set.

\(^{164}\) Such as an integer type, with constant integer values (6.7.1).
The following terms apply to objects and values of bitmask types:

- To set a value \( Y \) in an object \( X \) is to evaluate the expression \( X |\!\!= Y \).
- To clear a value \( Y \) in an object \( X \) is to evaluate the expression \( X &\!\!= \sim Y \).
- The value \( Y \) is set in the object \( X \) if the expression \( X & Y \) is nonzero.

### 20.4.2.1.5 Character sequences

The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

- A letter is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.
- The decimal-point character is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clause 21 through Clause 33 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*)`, or by a change to a `locale` object, as described in 25.3 and Clause 30.
- A character sequence is an array object (11.3.4) that can be declared as `T A[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.

### 20.4.2.1.5.1 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.
- 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.

### 20.4.2.1.5.2 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.
- A static NTMBS is an NTMBS with static storage duration.

### 20.4.2.2 Functions within classes

For the sake of exposition, Clause 21 through Clause 33 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 (15.1, 15.4, 15.8). It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

For the sake of exposition, the library clauses sometimes annotate constructors with `EXPLICIT`. Such a constructor is conditionally declared as either explicit or non-explicit (15.3.1). [Note: This is typically implemented by declaring two such constructors, of which at most one participates in overload resolution. — end note]
20.4.2.3 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.

```cpp
bool operator!=(const T& x, const T& y);
  Requires: Type \( T \) is EqualityComparable (Table 20).
  Returns: !(x == y).

bool operator>(const T& x, const T& y);
  Requires: Type \( T \) is LessThanComparable (Table 21).
  Returns: y < x.

bool operator<=(const T& x, const T& y);
  Requires: Type \( T \) is LessThanComparable (Table 21).
  Returns: !(y < x).

bool operator>=(const T& x, const T& y);
  Requires: Type \( T \) is LessThanComparable (Table 21).
  Returns: !(x < y).
```

20.4.2.4 Private members

Clause 21 through Clause 33 and Annex D do not specify the representation of classes, and intentionally omit specification of class members (12.2). 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 21 through Clause 33 and Annex D.

For the sake of exposition, some subclauses provide representative declarations, and semantic requirements, for private members of classes that meet the external specifications of the classes. The declarations for such members are followed by a comment that ends with `exposition only`, as in:

```cpp
streambuf* sb; // exposition only
```

An implementation may use any technique that provides equivalent observable behavior.

20.5 Library-wide requirements

This subclause specifies requirements that apply to the entire C++ standard library. Clause 21 through Clause 33 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, 20.5.1 describes the library’s contents and organization, 20.5.2 describes how well-formed C++ programs gain access to library entities, 20.5.3 describes constraints on types and functions used with the C++ standard library, 20.5.4 describes constraints on well-formed C++ programs, and 20.5.5 describes constraints on conforming implementations.

20.5.1 Library contents and organization

20.5.1.1 describes the entities and macros defined in the C++ standard library. 20.5.1.2 lists the standard library headers and some constraints on those headers. 20.5.1.3 lists requirements for a freestanding implementation of the C++ standard library.

20.5.1.1 Library contents

The C++ standard library provides definitions for the entities and macros described in the synopses of the C++ standard library headers (20.5.1.2).

All library entities except `operator new` and `operator delete` are defined within the namespace `std` or namespaces nested within namespace `std`.\(^{169}\) It is unspecified whether names declared in a specific namespace are declared directly in that namespace or in an inline namespace inside that namespace.\(^{170}\)

\(^{169}\) The C standard library headers (D.5) also define names within the global namespace, while the C++ headers for C library facilities (20.5.1.2) may also define names within the global namespace.

\(^{170}\) This gives implementers freedom to use inline namespaces to support multiple configurations of the library.
3 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.

20.5.1.2 Headers

1 Each element of the C++ standard library is declared or defined (as appropriate) in a header.

2 The C++ standard library provides the C++ library headers, shown in Table 16.

Table 16 — C++ library headers

| <algorithm> | <fstream> | <mutex> | <string> |
| <any> | <functional> | <new> | <string_view> |
| <array> | <future> | <numeric> | <strstream> |
| <atomic> | <initializer_list> | <optional> | <sysncstream> |
| <bitset> | <iostream> | <ostream> | <system_error> |
| <charconv> | <ios> | <queue> | <thread> |
| <chrono> | <iosfwd> | <random> | <tuple> |
| <codecvt> | <iosstream> | <ratio> | <type_traits> |
| <compare> | <iostream> | <regex> | <typeindex> |
| <complex> | <iterator> | <scoped_allocator> | <typeinfo> |
| <condition_variable> | <limits> | <set> | <unordered_map> |
| <deque> | <list> | <shared_mutex> | <unordered_set> |
| <exception> | <locale> | <sstream> | <utility> |
| <execution> | <map> | <stack> | <valarray> |
| <filesystem> | <memory> | <stdexcept> | <variant> |
| <forward_list> | <memory_resource> | <streambuf> | <vector> |

3 The facilities of the C standard library are provided in the additional headers shown in Table 17.

Table 17 — C++ headers for C library facilities

| <cassert> | <cinttypes> | <csignal> | <cstdio> | <cwchar> |
| <ccomplex> | <ciso646> | <cstdalign> | <cstdlib> | <cwctype> |
| <cctype> | <climits> | <cstdarg> | <cstring> |
| <cerrno> | <clocale> | <cstdlib> | <ctgmath> |
| <cfenv> | <cmath> | <cstddef> | <ctime> |
| <cfloat> | <cstdlib> | <ctime> | <vector> |

4 Except as noted in Clause 20 through Clause 33 and Annex D, the contents of each header c name 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 21 through Clause 33 and Annex D) are first declared within the global namespace scope and are then injected into namespace std by explicit using-declarations (10.3.3).

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.

171) A header is not necessarily a source file, nor are the sequences delimited by < and > in header names necessarily valid source file names (19.2).

172) It is intentional that there is no C++ header for any of these C headers: <statatomic.h>, <stdnoreturn.h>, <threads.h>.

173) 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.

174) In particular, including the standard header <ciso646.h> or <ciso646.h> has no effect.
D.5, C standard library headers, describes the effects of using the name .h (C header) form in a C++ program.\footnote{8} Annex K of the C standard describes a large number of functions, with associated types and macros, which "promote safer, more secure programming" than many of the traditional C library functions. The names of the functions have a suffix of _s; most of them provide the same service as the C library function with the unsuffixed name, but generally take an additional argument whose value is the size of the result array. If any C++ header is included, it is implementation-defined whether any of these names is declared in the global namespace. (None of them is declared in namespace std.)\footnote{9} Table 18 lists the Annex K names that may be declared in some header. These names are also subject to the restrictions of 20.5.4.3.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function Name</th>
<th>Function Name</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort_handler_s</td>
<td>mbstowcs_s</td>
<td>strncat_s</td>
<td>vswscanf_s</td>
</tr>
<tr>
<td>asctime_s</td>
<td>memcpys</td>
<td>strncpy_s</td>
<td>vprintf_s</td>
</tr>
<tr>
<td>bsearch_s</td>
<td>memmove_s</td>
<td>strtok_s</td>
<td>vsprintf_s</td>
</tr>
<tr>
<td>constraint_handler_t</td>
<td>memset_s</td>
<td>swprintf_s</td>
<td>wctomb_s</td>
</tr>
<tr>
<td>ctime_s</td>
<td>printf_s</td>
<td>swscanf_s</td>
<td>wcscat_s</td>
</tr>
<tr>
<td>errno_t</td>
<td>qsort_s</td>
<td>tmpfile_s</td>
<td>wccpy_s</td>
</tr>
<tr>
<td>fopen_s</td>
<td>RSIZE_MAX</td>
<td>TMP_MAX_S</td>
<td>wcscat_s</td>
</tr>
<tr>
<td>fprintf_s</td>
<td>rsize_t</td>
<td>tmpnam_s</td>
<td>wscnpy_s</td>
</tr>
<tr>
<td>freopen_s</td>
<td>scanf_s</td>
<td>vsprintf_s</td>
<td>wstrnlen_s</td>
</tr>
<tr>
<td>fscanf_s</td>
<td>set_constraint_handler_s</td>
<td>vscanf_s</td>
<td>wstrnlen_s</td>
</tr>
<tr>
<td>fwprintf_s</td>
<td>snprintf_s</td>
<td>vfwprintf_s</td>
<td>wcstombs_s</td>
</tr>
<tr>
<td>fswscanf_s</td>
<td>snprintf_s</td>
<td>vfwscanf_s</td>
<td>wcstombs_s</td>
</tr>
<tr>
<td>getenv_s</td>
<td>sprintf_s</td>
<td>vfprintf_s</td>
<td>wcstomb_s</td>
</tr>
<tr>
<td>gets_s</td>
<td>sscanf_s</td>
<td>vsscanf_s</td>
<td>wcctype_s</td>
</tr>
<tr>
<td>gmtime_s</td>
<td>strcat_s</td>
<td>vsnprintf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>ignore_handler_s</td>
<td>strcPy_s</td>
<td>wcharprintf_s</td>
<td>wmemmove_s</td>
</tr>
<tr>
<td>L_tmpnam_s</td>
<td>strerror_s</td>
<td>vsprintf_s</td>
<td>wcharprintf_s</td>
</tr>
<tr>
<td>localtime_s</td>
<td>strerror_s</td>
<td>vsscanf_s</td>
<td>wcharprintf_s</td>
</tr>
<tr>
<td>mbsrtowcs_s</td>
<td>strlen_s</td>
<td>vswprintf_s</td>
<td>wcharprintf_s</td>
</tr>
</tbody>
</table>

20.5.1.3 Freestanding implementations [compliance] Two kinds of implementations are defined: \textit{hosted} and \textit{freestanding} (4.1). 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 19.

The supplied version of the header \texttt{<cstdlib>} shall declare at least the functions \texttt{abort}, \texttt{atexit}, \texttt{at_quick_exit}, \texttt{exit}, and \texttt{quick_exit} (21.5). The other headers listed in this table shall meet the same requirements as for a hosted implementation.

20.5.2 Using the library [using]

20.5.2.1 Overview [using.overview]

Subclause 20.5.2 describes how a C++ program gains access to the facilities of the C++ standard library. 20.5.2.2 describes effects during translation phase 4, while 20.5.2.3 describes effects during phase 8 (5.2).

20.5.2.2 Headers [using.headers]

The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate \#include preprocessing directive (19.2).

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 \texttt{<cassert>} or \texttt{<cassert.h>} depends each time on the lexically current definition of \texttt{NDEBUG}.\footnote{176}
Table 19 — C++ headers for freestanding implementations

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2 Types</td>
<td><code>&lt;ciso646&gt;</code></td>
</tr>
<tr>
<td>21.3 Implementation properties</td>
<td><code>&lt;cstdint&gt;</code></td>
</tr>
<tr>
<td>21.4 Integer types</td>
<td><code>&lt;cfloat&gt;</code> <code>&lt;limits&gt;</code> <code>&lt;climits&gt;</code></td>
</tr>
<tr>
<td>21.5 Start and termination</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>21.6 Dynamic memory management</td>
<td><code>&lt;new&gt;</code></td>
</tr>
<tr>
<td>21.7 Type identification</td>
<td><code>&lt;typeinfo&gt;</code></td>
</tr>
<tr>
<td>21.8 Exception handling</td>
<td><code>&lt;exception&gt;</code></td>
</tr>
<tr>
<td>21.9 Initializer lists</td>
<td><code>&lt;initializer_list&gt;</code></td>
</tr>
<tr>
<td>21.11 Other runtime support</td>
<td><code>&lt;cstdlib&gt;</code></td>
</tr>
<tr>
<td>23.15 Type traits</td>
<td><code>&lt;type_traits&gt;</code></td>
</tr>
<tr>
<td>Clause 32 Atomics</td>
<td><code>&lt;atomic&gt;</code></td>
</tr>
<tr>
<td>D.4.2, D.4.3 Deprecated headers</td>
<td><code>&lt;cstdbool&gt;</code> <code>&lt;cstdalign&gt;</code> <code>&lt;cstdbool&gt;</code> <code>&lt;cstdbool&gt;</code></td>
</tr>
</tbody>
</table>

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.

20.5.2.3 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 (10.5).

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.\(^{177}\)

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 (20.5.4.6), runtime changes (20.5.4.7).

20.5.3 Requirements on types and expressions

1 20.5.3.1 describes requirements on types and expressions used to instantiate templates defined in the C++ standard library. 20.5.3.2 describes the requirements on swappable types and swappable expressions. 20.5.3.3 describes the requirements on pointer-like types that support null values. 20.5.3.4 describes the requirements on hash function objects. 20.5.3.5 describes the requirements on storage allocators.

20.5.3.1 Template argument requirements

1 The template definitions in the C++ standard library refer to various named requirements whose details are set out in Tables 20–27. 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 (11.6) if one of those signatures is called using the default argument (11.3.6).\(^{177}\)

\(^{177}\) 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 20 — EqualityComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
</table>
| `a == b`   | convertible to `bool` | `==` is an equivalence relation, that is, it has the following properties:  
  — For all `a`, `a == a`.  
  — If `a == b`, then `b == a`.  
  — If `a == b` and `b == c`, then `a == c`. |

Table 21 — LessThanComparable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &lt; b</code></td>
<td>convertible to <code>bool</code></td>
<td><code>&lt;</code> is a strict weak ordering relation (28.7)</td>
</tr>
</tbody>
</table>

Table 22 — DefaultConstructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T t;</code></td>
<td>object <code>t</code> is default-initialized</td>
</tr>
<tr>
<td><code>T u{}</code></td>
<td>object <code>u</code> is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td><code>T()</code></td>
<td>an object of type <code>T</code> is value-initialized or aggregate-initialized</td>
</tr>
<tr>
<td>`T{}</td>
<td></td>
</tr>
</tbody>
</table>

Table 23 — MoveConstructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T u = rv;</code></td>
<td><code>u</code> is equivalent to the value of <code>rv</code> before the construction</td>
</tr>
<tr>
<td><code>T(rv)</code></td>
<td><code>T(rv)</code> is equivalent to the value of <code>rv</code> before the construction</td>
</tr>
<tr>
<td><code>rv</code>'s state is unspecified</td>
<td>[Note: <code>rv</code> must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether <code>rv</code> has been moved from or not. — end note]</td>
</tr>
</tbody>
</table>

Table 24 — CopyConstructible requirements (in addition to MoveConstructible)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T u = v;</code></td>
<td>the value of <code>v</code> is unchanged and is equivalent to <code>u</code></td>
</tr>
<tr>
<td><code>T(v)</code></td>
<td>the value of <code>v</code> is unchanged and is equivalent to <code>T(v)</code></td>
</tr>
</tbody>
</table>

Table 25 — MoveAssignable requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Return value</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>t = rv</code></td>
<td><code>T&amp;</code></td>
<td><code>t</code></td>
<td>If <code>t</code> and <code>rv</code> do not refer to the same object, <code>t</code> is equivalent to the value of <code>rv</code> before the assignment</td>
</tr>
<tr>
<td><code>rv</code>'s state is unspecified</td>
<td>[Note: <code>rv</code> must still meet the requirements of the library component that is using it, whether or not <code>t</code> and <code>rv</code> refer to the same object. The operations listed in those requirements must work as specified whether <code>rv</code> has been moved from or not. — end note]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 26 — CopyAssignable requirements (in addition to MoveAssignable)

<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 27 — Destructible requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>u.~T()</td>
<td>All resources owned by u are reclaimed, no exception is propagated.</td>
</tr>
</tbody>
</table>

20.5.3.2 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 swap(t, u) and 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 swap(t, u) and swap(u, t) are evaluated shall ensure that a binary non-member function named “swap” is selected via overload resolution (16.3) on a candidate set that includes:

1. The two swap function templates defined in <utility> (23.2) 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 std::swap(t, u) or 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 (27.2) satisfies the requirements of ValueSwappable if, for any dereferenceable object x of type X, *x is swappable.

[Example: User code can ensure that the evaluation of swap calls is performed in an appropriate context under the various conditions as follows:

```c++
#include <utility>

// Requires: std::forward<T>(t) shall be swappable with std::forward<U>(u).
template<class T, class U>
void value_swap(T&& t, U&& u) {
    using std::swap;
    swap(std::forward<T>(t), std::forward<U>(u)); // OK: uses “swappable with” conditions
    // for rvalues and lvalues
}

// Requires: lvalues of T shall be swappable.
template<class T>
void lv_swap(T& t1, T& t2) {
    using std::swap;
    swap(t1, t2); // OK: uses swappable conditions for lvalues of type T
}
```]
namespace N {
  struct A { int m; }
  struct Proxy { A* a; }
  Proxy proxy(A& a) { return Proxy{ a }; }

  void swap(A& x, Proxy p) {
    std::swap(x.m, p.a->m);
    // OK: uses context equivalent to swappable
    // 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

20.5.3.3 NullablePointer requirements

A NullablePointer type is a pointer-like type that supports null values. A type \( P \) meets the requirements of NullablePointer if:

1. \( P \) satisfies the requirements of EqualityComparable, DefaultConstructible, CopyConstructible, CopyAssignable, and Destructible,
2. values of type \( P \) are swappable (20.5.3.2),
3. the expressions shown in Table 28 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 (Clause 7). The effect shall be as if \( p != nullptr \) had been evaluated in place of \( p \).

No operation which is part of the NullablePointer requirements shall exit via an exception.

In Table 28, \( u \) 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 28 — NullablePointer 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>Postconditions: ( u == nullptr )</td>
</tr>
<tr>
<td>( P u = np; )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( P(np) )</td>
<td></td>
<td>Postconditions: ( P(np) == nullptr )</td>
</tr>
<tr>
<td>( t = np P&amp; )</td>
<td></td>
<td>Postconditions: ( t == nullptr )</td>
</tr>
<tr>
<td>( a != b )</td>
<td>contextually convertible to bool</td>
<td>( !(a == b) )</td>
</tr>
<tr>
<td>( a == np )</td>
<td>contextually convertible to bool</td>
<td>( a == P() )</td>
</tr>
<tr>
<td>( np == a )</td>
<td>contextually convertible to bool</td>
<td>( a == np )</td>
</tr>
<tr>
<td>( a != np )</td>
<td>contextually convertible to bool</td>
<td>( !(a == np) )</td>
</tr>
</tbody>
</table>

20.5.3.4 Hash requirements

A type \( H \) meets the Hash requirements if:

§ 20.5.3.4 421
— it is a function object type (23.14),
— it satisfies the requirements of CopyConstructible and Destructible (20.5.3.1), and
— the expressions shown in Table 29 are valid and have the indicated semantics.

Given Key is an argument type for function objects of type H, in Table 29 h is a value of type (possibly const) H, u is an lvalue of type Key, and k is a value of a type convertible to (possibly const) Key.

Table 29 — Hash 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>

20.5.3.5 Allocator requirements

The library describes a standard set of requirements for allocators, which are class-type objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for it. All of the string types (Clause 24), containers (Clause 26) (except array), string buffers and string streams (Clause 30), and match_results (Clause 31) are parameterized in terms of allocators.

The class template allocator_traits (23.10.9) supplies a uniform interface to all allocator types. Table 30 describes the types manipulated through allocators. Table 31 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 31 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 30 and 31, the use of move and forward always refers to std::move and std::forward, respectively.

Table 30 — 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 allocator_traits&lt;X&gt;</td>
</tr>
<tr>
<td>YY</td>
<td>the type allocator_traits&lt;Y&gt;</td>
</tr>
<tr>
<td>a, a1, a2</td>
<td>lvalues of type X</td>
</tr>
<tr>
<td>u</td>
<td>the name of a variable being declared</td>
</tr>
<tr>
<td>b</td>
<td>a value of type Y</td>
</tr>
<tr>
<td>c</td>
<td>a pointer of type C* through which indirection is valid</td>
</tr>
<tr>
<td>p</td>
<td>a value of type XX::pointer, obtained by calling a1.allocate, where a1 == a</td>
</tr>
<tr>
<td>q</td>
<td>a value of type XX::const_pointer obtained by conversion from a value p.</td>
</tr>
<tr>
<td>r</td>
<td>a value of type T* 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>
</tbody>
</table>
Table 30 — Descriptive variable definitions (continued)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<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 31 — Allocator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/post-condition</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::pointer</td>
<td>X::pointer</td>
<td>X::pointer is convertible to X::const_pointer</td>
<td>pointer_traits&lt;X::pointer&gt;::rebind&lt;const T&gt;</td>
</tr>
<tr>
<td>X::const_pointer</td>
<td>X::pointer</td>
<td>X::const_pointer</td>
<td></td>
</tr>
<tr>
<td>X::void_pointer</td>
<td>X::pointer</td>
<td>X::const_pointer</td>
<td>pointer_traits&lt;X::pointer&gt;::rebind&lt;void&gt;</td>
</tr>
<tr>
<td>Y::void_pointer</td>
<td>X::pointer</td>
<td>X::const_pointer</td>
<td>pointer_traits&lt;X::pointer&gt;::rebind&lt;void&gt;</td>
</tr>
<tr>
<td>X::const_void_pointer</td>
<td>X::pointer</td>
<td>X::const_pointer</td>
<td>pointer_traits&lt;X::pointer&gt;::rebind&lt;void&gt;</td>
</tr>
<tr>
<td>Y::const_void_pointer</td>
<td>X::pointer</td>
<td>X::const_pointer</td>
<td>pointer_traits&lt;X::pointer&gt;::rebind&lt;void&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X::value_type</th>
<th>Identical to T</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::size_type</td>
<td>unsigned integer type</td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integer type</td>
</tr>
</tbody>
</table>

| typename X::template rebind<U>::other | Y | For all U (including T), Y::template rebind<T>::other is X. | See Note A, below. |

| *p | T& |
| *q | const T& |
| p->m | type of T::m |
| q->m | type of T::m |

Requires: (*p).m is well-defined. equivalent to (*p).m

Requires: (*q).m is well-defined. equivalent to (*q).m

§ 20.5.3.5
Table 31 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default</th>
</tr>
</thead>
</table>
| static_cast<
X::pointer>(w) | X::pointer | static_cast<X::pointer>(w) == p | |
| static_cast<
X::const_pointer
>(x) | X::const_pointer | static_cast<X::const_pointer>(x) == q | |
| pointer_traits<
X::pointer
::pointer_to(r) | X::pointer | same as p | |
| a.allocate(n) | X::pointer | Memory is allocated for n objects of type T but objects are not constructed. allocate may throw an appropriate exception.[Note: If n == 0, the return value is unspecified. —end note] | |
| a.allocate(n, y) | X::pointer | Same as a.allocate(n). The use of y is unspecified, but it is intended as an aid to locality. | |
| a.deallocate(p, n) | (not used) | 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. | |
| a.max_size() | X::size_type | the largest value that can meaningfully be passed to X::allocate() | numeric_limits<
size_type>::max() / sizeof(value_type) |
| a1 == a2 | bool | returns true only if storage allocated from each can be deallocated via the other. operator== shall be reflexive, symmetric, and transitive, and shall not exit via an exception. | |
| a1 != a2 | bool | same as !(a1 == a2) | |
| a == b | bool | same as a == Y::rebind<T>::other(b) | |
| a != b | bool | same as !(a == b) | |
| X u(a); | Shall not exit via an exception. Postconditions: u == a | |
| X u = a; | | |
| X u(b); | Shall not exit via an exception. Postconditions: Y(u) == b, u == X(b) | |
| X u(std::move(a)); | Shall not exit via an exception. Postconditions: The value of a is unchanged and is equal to u. | |

178 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 31 — Allocator requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Default pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>X ( u(\text{std::move}(b)); )</td>
<td></td>
<td>Shall not exit via an exception. Postconditions: ( u ) is equal to the prior value of ( X(b) ).</td>
<td></td>
</tr>
<tr>
<td>a.construct(c, args)</td>
<td>(not used)</td>
<td>Effects: Constructs an object of type ( C ) at ( c ) (:=\text{new}((\text{void}*)c)) ( C(\text{forward}&lt;\text{T}\text{Arg}&lt;\text{Args}&gt;\text{args}))…</td>
<td></td>
</tr>
<tr>
<td>a.destroy(c)</td>
<td>(not used)</td>
<td>Effects: Destroys the object at ( c ) ( c-&gt;\sim\text{C}() )</td>
<td></td>
</tr>
<tr>
<td>a.select_on_container_copy_construction()</td>
<td>( X )</td>
<td>Typically returns either ( a ) or ( X() ) return ( a );</td>
<td></td>
</tr>
<tr>
<td>( X::\text{propagate}<em>{\text{on}}</em>{\text{container}}<em>{\text{copy}}</em>{\text{assignment}} )</td>
<td>Identical to or derived from ( \text{true_type} ) or ( \text{false_type} )</td>
<td>( \text{true_type} ) only if an allocator of type ( X ) should be copied when the client container is copy-assigned. See Note B, below.</td>
<td></td>
</tr>
<tr>
<td>( X::\text{propagate}<em>{\text{on}}</em>{\text{container}}<em>{\text{move}}</em>{\text{assignment}} )</td>
<td>Identical to or derived from ( \text{true_type} ) or ( \text{false_type} )</td>
<td>( \text{true_type} ) only if an allocator of type ( X ) should be moved when the client container is move-assigned. See Note B, below.</td>
<td></td>
</tr>
<tr>
<td>( X::\text{propagate}<em>{\text{on}}</em>{\text{container}}_{\text{swap}} )</td>
<td>Identical to or derived from ( \text{true_type} ) or ( \text{false_type} )</td>
<td>( \text{true_type} ) only if an allocator of type ( X ) should be swapped when the client container is swapped. See Note B, below.</td>
<td></td>
</tr>
<tr>
<td>( X::\text{is_always_equal} )</td>
<td>Identical to or derived from ( \text{true_type} ) or ( \text{false_type} )</td>
<td>( \text{true_type} ) only if the expression ( a1 == a2 ) is guaranteed to be true for any two (possibly ( \text{const} )) values ( a1, a2 ) of type ( X ).</td>
<td></td>
</tr>
</tbody>
</table>

3 Note A: The member class template \( \text{rebind} \) in the table above is effectively a typedef template. [Note: In general, if the name \( \text{Allocator} \) is bound to \( \text{SomeAllocator}\langle\text{T}\rangle \), then \( \text{Allocator}::\text{rebind}\langle\text{U}\rangle::\text{other} \) is the same type as \( \text{SomeAllocator}\langle\text{U}\rangle \), where \( \text{SomeAllocator}\langle\text{T}\rangle::\text{value\_type} \) is \( \text{T} \) and \( \text{SomeAllocator}\langle\text{U}\rangle::\text{value\_type} \) is \( \text{U} \). —end note] If \( \text{Allocator} \) is a class template instantiation of the form \( \text{SomeAllocator}\langle\text{T}\rangle\langle\text{Arg}<\text{Args}>\rangle \), where \( \text{Args} \) is zero or more type arguments, and \( \text{Allocator} \) does not supply a \( \text{rebind} \) member template, the standard \( \text{allocator\_traits} \) template uses \( \text{SomeAllocator}\langle\text{U}, \text{Args}\rangle \) in place of \( \text{Allocator}::\text{rebind}\langle\text{U}\rangle::\text{other} \) by default. For allocator types that are not template instantiations of the above form, no default is provided.

4 Note B: If \( X::\text{propagate}_{\text{on}}_{\text{container}}_{\text{copy}}_{\text{assignment}}::\text{value} \) is \( \text{true} \), \( X \) shall satisfy the \( \text{CopyAssignable} \) requirements (Table 26) and the copy operation shall not throw exceptions. If \( X::\text{propagate}_{\text{on}}_{\text{container}}_{\text{move}}_{\text{assignment}}::\text{value} \) is \( \text{true} \), \( X \) shall satisfy the \( \text{MoveAssignable} \) requirements (Table 25) and the move operation shall not throw exceptions. If \( X::\text{propagate}_{\text{on}}_{\text{container}}_{\text{swap}}::\text{value} \) is \( \text{true} \), values of type \( X \) shall be swappable (20.5.3.2) and the swap operation shall not throw exceptions.

5 An allocator type \( X \) shall satisfy the requirements of \( \text{CopyConstructible} \) (20.5.3.1). The \( X::\text{pointer}, X::\text{const\_pointer}, X::\text{void\_pointer}, \) and \( X::\text{const\_void\_pointer} \) types shall satisfy the requirements of \( \text{NullablePointer} \) (20.5.3.3). No constructor, comparison function, copy operation, move operation, or swap operation on these pointer types shall exit via an exception. \( X::\text{pointer} \) and \( X::\text{const\_pointer} \) shall also satisfy the requirements for a random access iterator (27.2.7) and of a contiguous iterator (27.2.1).

6 Let \( x1 \) and \( x2 \) denote objects of (possibly different) types \( X::\text{void\_pointer}, X::\text{const\_void\_pointer}, X::\text{pointer}, \) or \( X::\text{const\_pointer} \). Then, \( x1 \) and \( x2 \) are \( \text{equivalently\_valued} \) 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.

7 Let w1 and w2 denote objects of type X::void_pointer. Then for the expressions

\[
w_1 == w2
\]
\[
w_1 != w2
\]

either or both objects may be replaced by an equivalently-valued object of type X::const_void_pointer with no change in semantics.

8 Let p1 and p2 denote objects of type X::pointer. Then for the expressions

\[
p1 == p2
\]
\[
p1 != p2
\]
\[
p1 < p2
\]
\[
p1 <= p2
\]
\[
p1 > p2
\]
\[
p1 >= p2
\]
\[
p1 - p2
\]

either or both objects may be replaced by an equivalently-valued object of type X::const_pointer with no change in semantics.

9 An allocator may constrain the types on which it can be instantiated and the arguments for which its construct or destroy members may be called. If a type cannot be used with a particular allocator, the allocator class or the call to construct or destroy may fail to instantiate.

[Example: The following is an allocator class template supporting the minimal interface that satisfies the requirements of Table 31:

```cpp
template<class Tp>
struct SimpleAllocator {
    typedef Tp value_type;
    SimpleAllocator(ctor args);
    template<class T> SimpleAllocator(const SimpleAllocator<T>& other);
    [[nodiscard]] Tp* allocate(std::size_t n);
    void deallocate(Tp* p, std::size_t n);
};

template<class T, class U>
bool operator==(const SimpleAllocator<T>&, const SimpleAllocator<U>&);
```
—end example]

10 If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment. [Note: Additionally, the member function allocate for that type may fail by throwing an object of type bad_alloc. —end note]

20.5.4 Constraints on programs  

20.5.4.1 Overview  

Subclause 20.5.4 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 (20.5.4.2.1), its use of various reserved names (20.5.4.3), its use of headers (20.5.4.4), its use of standard library classes as base
classes (20.5.4.5), its definitions of replacement functions (20.5.4.6), and its installation of handler functions
during execution (20.5.4.7).

20.5.4.2 Namespace use
20.5.4.2.1 Namespace std

1 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 otherwise specified. A program may add a template specialization
for any standard library template to namespace std only if the declaration depends on a user-defined type
and the specialization meets the standard library requirements for the original template and is not explicitly
prohibited.\footnote{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.}

2 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.

3 The behavior of a C++ program is undefined if it declares

\begin{itemize}
\item an explicit specialization of any member function of a standard library class template, or
\item an explicit specialization of any member function template of a standard library class or class template,
\item an explicit or partial specialization of any member class template of a standard library class or class
template, or
\item a deduction guide for any standard library class template.
\end{itemize}

A program may explicitly instantiate a template defined in the standard library only if the declaration
depends on the name of a user-defined type and the instantiation meets the standard library requirements
for the original template.

4 A translation unit shall not declare namespace std to be an inline namespace (10.3.1).

20.5.4.2.2 Namespace posix

1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace posix or to a
namespace within namespace posix unless otherwise specified. The namespace posix is reserved for use by
ISO/IEC 9945 and other POSIX standards.

20.5.4.2.3 Namespaces for future standardization

1 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]

20.5.4.3 Reserved names

1 The C++ standard library reserves the following kinds of names:

\begin{itemize}
\item macros
\item global names
\item names with external linkage
\end{itemize}

2 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.

20.5.4.3.1 Zombie names

1 In namespace std, the following names are reserved for previous standardization:

\begin{itemize}
\item auto_ptr,
\item binary_function,
\item bind1st,
\item bind2nd,
\item binder1st,
\end{itemize}
20.5.4.3.2 Macro names

1 A translation unit that includes a standard library header shall not define or undef names declared in any standard library header.

2 A translation unit shall not define or undef names lexically identical to keywords, to the identifiers listed in Table 4, or to the attribute-tokens described in 10.6.

20.5.4.3.3 External linkage

1 Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage, both in namespace std and in the global namespace.

2 Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.

3 Each name from the C standard library declared with external linkage is reserved to the implementation for use as a name with extern "C" linkage, both in namespace std and in the global namespace.

4 Each function signature from the C standard library declared with external linkage is reserved to the implementation for use as a function signature with both extern "C" and extern "C++" linkage, or as a name of namespace scope in the global namespace.

20.5.4.3.4 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.

20.5.4.3.5 User-defined literal suffixes

1 Literal suffix identifiers (16.5.8) that do not start with an underscore are reserved for future standardization.

---

180) The list of such reserved names includes errno, declared or defined in <cerrno>.  
181) The list of such reserved function signatures with external linkage includes setjmp(jmp_buf), declared or defined in <setjmp>, and va_end(va_list), declared or defined in <stdarg>.  
182) The function signatures declared in <uchar>, <wchar>, and <cwctype> are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.  
183) These types are clock_t, div_t, FILE, fpos_t, lconv, ldiv_t, mbstate_t, ptdiff_t, sig_atomic_t, size_t, time_t, tm, va_list, wctrans_t, wctype_t, and wint_t.
20.5.4.4 Headers

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 (19.2), the behavior is undefined.

20.5.4.5 Derived classes

Virtual member function signatures defined for a base class in the C++ standard library may be overridden in a derived class defined in the program (13.3).

20.5.4.6 Replacement functions

Clause 21 through Clause 33 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 (20.3).

A C++ program may provide the definition for any of the following dynamic memory allocation function signatures declared in header <new> (6.6.4.4, 21.6):

operator new(std::size_t)
operator new(std::size_t, std::align_val_t)
operator new(std::size_t, const std::nothrow_t&)
operator new(std::size_t, std::align_val_t, const std::nothrow_t&)
operator delete(void*)
operator delete(void*, std::size_t)
operator delete(void*, std::align_val_t)
operator delete(void*, std::size_t, std::align_val_t)
operator delete(void*, const std::nothrow_t&)
operator delete(void*, std::align_val_t, const std::nothrow_t&)
operator new[](std::size_t)
operator new[](std::size_t, std::align_val_t)
operator new[](std::size_t, const std::nothrow_t&)
operator new[](std::size_t, std::align_val_t, const std::nothrow_t&)
operator delete[](void*)
operator delete[](void*, std::size_t)
operator delete[](void*, std::align_val_t)
operator delete[](void*, std::size_t, std::align_val_t)
operator delete[](void*, const std::nothrow_t&)
operator delete[](void*, std::align_val_t, const std::nothrow_t&)

The program’s definitions are used instead of the default versions supplied by the implementation (21.6). 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.

20.5.4.7 Handler functions

The C++ standard library provides a default version of the following handler function (Clause 21):

(1.1) — terminate_handler

(2.1) — set_new_handler
(2.2) — set_terminate

See also subclauses 21.6.3, Storage allocation errors, and 21.8, 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.
20.5.4.8 Other functions

In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, this document places no requirements on the implementation.

In particular, the effects are undefined in the following cases:

1. For replacement functions (21.6.2), if the installed replacement function does not implement the semantics of the applicable Required behavior: paragraph.
2. For handler functions (21.6.3.3, 21.8.4.1), if the installed handler function does not implement the semantics of the applicable Required behavior: paragraph.
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 (20.5.3.5, 26.2, 27.2, 28.3, 29.3). Operations on such types can report a failure by throwing an exception unless otherwise specified.
4. If any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable Required behavior: paragraph.
5. If an incomplete type (6.7) is used as a template argument when instantiating a template component, unless specifically allowed for that component.

20.5.4.9 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 (17.9.2.1) 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]

20.5.4.10 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 20.5.5.9. [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]

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]

20.5.4.11 Requires paragraph

Violation of the preconditions specified in a function’s Requires: paragraph results in undefined behavior unless the function’s Throws: paragraph specifies throwing an exception when the precondition is violated.

20.5.5 Conforming implementations

20.5.5.1 Overview

Subclause 20.5.5 describes the constraints upon, and latitude of, implementations of the C++ standard library.
An implementation’s use of headers is discussed in 20.5.5.2, its use of macros in 20.5.5.3, non-member functions in 20.5.5.4, member functions in 20.5.5.5, data race avoidance in 20.5.5.9, access specifiers in 20.5.5.10, class derivation in 20.5.5.11, and exceptions in 20.5.5.12.

20.5.5.2 Headers

A C++ header may include other C++ headers. A C++ header shall provide the declarations and definitions that appear in its synopsis. A C++ header shown in its synopsis as including other C++ headers shall provide the declarations and definitions that appear in the synopses of those other headers.

Certain types and macros are defined in more than one header. Every such entity shall be defined such that any header that defines it may be included after any other header that also defines it (6.2).

The C standard library headers (D.5) shall include only their corresponding C++ standard library header, as described in 20.5.1.2.

20.5.5.3 Restrictions on macro definitions

The names and global function signatures described in 20.5.1.1 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.

20.5.5.4 Non-member functions

A call to a non-member function signature described in Clause 21 through Clause 33 and Annex D shall behave as if the implementation declared no additional non-member function signatures. 184

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 (20.5.3.2). The exception for overloaded operators allows argument-dependent lookup in cases like that of ostream_iterator::operator= (27.6.2.2): ]

Effects:

*out_stream << value;
if (delim != 0)
  *out_stream << delim;
return *this;
—end note]

20.5.5.5 Member functions

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]

20.5.5.6 Constexpr functions and constructors

This document explicitly requires that certain standard library functions are constexpr (10.1.5). 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.

184) 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.
20.5.5.7 Requirements for stable algorithms

When the requirements for an algorithm state that it is “stable” without further elaboration, it means:

1. For the sort algorithms the relative order of equivalent elements is preserved.
2. For the remove and copy algorithms the relative order of the elements that are not removed is preserved.
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).

20.5.5.8 Reentrancy

Except where explicitly specified in this document, it is implementation-defined which functions in the C++ standard library may be recursively reentered.

20.5.5.9 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.

1. 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.
2. 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.
3. 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.
4. 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.

20.5.5.10 Protection within classes

It is unspecified whether any function signature or class described in Clause 21 through Clause 33 and Annex D is a friend of another class in the C++ standard library.

20.5.5.11 Derived classes

An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

1. 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.
2. In any case:
   1. Every base class described as virtual shall be virtual;
   2. Every base class not specified as virtual shall not be virtual;
Unless explicitly stated otherwise, types with distinct names shall be distinct types.\(^{185}\)

All types specified in the C++ standard library shall be non-final types unless otherwise specified.

20.5.5.12 Restrictions on exception handling \(\text{[res.on.exception.handling]}\)

Any of the functions defined in the C++ standard library can report a failure by throwing an exception of a type described in its Throws: paragraph, or of a type derived from a type named in the Throws: paragraph that would be caught by an exception handler for the base type.

Functions from the C standard library shall not throw exceptions\(^{186}\) except when such a function calls a program-supplied function that throws an exception.\(^{187}\)

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.\(^{188}\) Implementations should report errors by throwing exceptions of or derived from the standard exception classes (21.6.3.1, 21.8, 22.2).

An implementation may strengthen the exception specification for a non-virtual function by adding a non-throwing exception specification.

20.5.5.13 Restrictions on storage of pointers \(\text{[res.on.pointer.storage]}\)

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 (6.6.4.4.3). \(\text{[Note: Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. Libraries that store pointers outside the user’s address space should make it appear that they are stored and retrieved from a traceable pointer location. — end note]}\)

20.5.5.14 Value of error codes \(\text{[value.error.codes]}\)

Certain functions in the C++ standard library report errors via a std::error_code (22.5.3.1) 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. \(\text{[Example: For operating systems that are based on POSIX, implementations should define the std::system_category()}\) 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]}\)

20.5.5.15 Moved-from state of library types \(\text{[lib.types.movedfrom]}\)

Objects of types defined in the C++ standard library may be moved from (15.8). 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.

\(^{185}\) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as size_t (21.2) and streamoff (30.5.2).

\(^{186}\) 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.

\(^{187}\) The functions qsort() and bsearch() (28.8) meet this condition.

\(^{188}\) 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 (21.6.3.1).
21 Language support library
[language.support]

21.1 General
[support.general]

This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 32.

Table 32 — Language support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2 Common definitions</td>
<td>&lt;cstdlib&gt; &lt;cstddef&gt;</td>
</tr>
<tr>
<td>21.3 Implementation properties</td>
<td>&lt;climits&gt; &lt;limits&gt; &lt;cfloat&gt;</td>
</tr>
<tr>
<td>21.4 Integer types</td>
<td>&lt;cstdint&gt;</td>
</tr>
<tr>
<td>21.5 Start and termination</td>
<td>&lt;cstdlib&gt;</td>
</tr>
<tr>
<td>21.6 Dynamic memory management</td>
<td>&lt;new&gt;</td>
</tr>
<tr>
<td>21.7 Type identification</td>
<td>&lt;typeinfo&gt;</td>
</tr>
<tr>
<td>21.8 Exception handling</td>
<td>&lt;exception&gt;</td>
</tr>
<tr>
<td>21.9 Initializer lists</td>
<td>&lt;initializer_list&gt;</td>
</tr>
<tr>
<td>21.10 Comparisons</td>
<td>&lt;compare&gt;</td>
</tr>
<tr>
<td>21.11 Other runtime support</td>
<td>&lt;csignal&gt; &lt;csetjmp&gt; &lt;cstdarg&gt; &lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

21.2 Common definitions
[support.types]

21.2.1 Header <cstdlib> synopsis
[csndef.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 {};

    // 21.2.5, byte type operations
    template<class IntType>
    constexpr byte operator<<(byte b, IntType shift) noexcept;
    template<class IntType>
    constexpr byte operator>>(byte b, IntType shift) noexcept;
    template<class IntType>
    constexpr byte operator|(byte b, IntType shift) noexcept;
    template<class IntType>
    constexpr byte operator|(byte b, IntType shift) noexcept;
    template<class IntType>
    constexpr byte operator|(byte b, IntType shift) noexcept;
    template<class IntType>
    constexpr byte operator|(byte b, IntType shift) noexcept;
}
constexpr byte& operator&=(byte& l, byte r) noexcept;
constexpr byte operator&(byte l, byte r) noexcept;
constexpr byte& operator^=(byte& l, byte r) noexcept;
constexpr byte operator^(byte l, byte r) noexcept;
constexpr byte operator ~(byte b) noexcept;
template<class IntType>
  constexpr IntType to_integer(byte b) noexcept;
}

#define NULL see below
#define offsetof(P, D) see below

1 The contents and meaning of the header `<cstdint>` are the same as the C standard library header `<cstdint.h>`, except that it does not declare the type wchar_t, that it also declares the type byte and its associated operations (21.2.5), and as noted in 21.2.3 and 21.2.4.

See also: ISO C 7.19
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);

// 24.5.6, multibyte / wide string and character conversion functions
int mbstrlen(const char* s, size_t n);
int mbtowcs(char* wpc, const char* s, size_t n);
int wcstombs(char* s, const wchar_t* wpc, size_t n);

// 28.8, 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);

// 29.6.9, low-quality random number generation
int rand();
void srand(unsigned int seed);

// 29.9.2, 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 int numer, long int denom); // see 20.2
ldiv_t div(long long int numer, long long int denom); // see 20.2
ldiv_t ldiv(long int numer, long int denom);
lldiv_t lldiv(long long int numer, long long int denom);

§ 21.2.3 Null pointers [support.types.nullptr]
The type nullptr_t is a synonym for the type nullptr expression, and it has the characteristics
described in 6.7.1 and 7.11. [Note: Although nullptr’s address cannot be taken, the address of another
nullptr_t object that is an lvalue can be taken. — end note]

See also: ISO C 7.19

21.2.4 Sizes, alignments, and offsets [support.types.layout]
The macro offsetof(type, member-designator) has the same semantics as the corresponding macro in
the C standard library header <stddef.h>, but accepts a restricted set of type arguments in this document.

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 21.2.3, 21.2.4, 21.5, 23.10.12, 24.5.6,
28.8, 28.6.9, and 29.9.2. [Note: Several functions have additional overloads in this document, but they have
the same behavior as in the C standard library (20.2). — end note]
Use of the `offsetof` macro with a `type` other than a standard-layout class (Clause 12) is conditionally-supported.\(^{190}\) The expression `offsetof(type, member-designator)` is never type-dependent (17.7.2.2) and it is value-dependent (17.7.2.3) 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 8.5.6.

The type `size_t` is an implementation-defined unsigned integer type that is large enough to contain the size in bytes of any object (8.5.2.3).

[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 type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context (6.6.5).

See also: ISO C 7.19

### 21.2.5  byte type operations

[support.types.byteops]

```cpp
template<class IntType>
constexpr byte& operator<<=(byte& b, IntType shift) noexcept;
1
   Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntType>` is true.
2
   Effects: Equivalent to: return b = b << shift;
```

```cpp
template<class IntType>
constexpr byte operator<<(byte b, IntType shift) noexcept;
3
   Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntType>` is true.
4
   Effects: Equivalent to:
   return static_cast<byte>(static_cast<unsigned char>(
      static_cast<unsigned int>(b) << shift));
```

```cpp
template<class IntType>
constexpr byte& operator>>=(byte& b, IntType shift) noexcept;
5
   Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntType>` is true.
6
   Effects: Equivalent to: return b >> shift;
```

```cpp
template<class IntType>
constexpr byte operator>>(byte b, IntType shift) noexcept;
7
   Remarks: This function shall not participate in overload resolution unless `is_integral_v<IntType>` is true.
8
   Effects: Equivalent to:
   return static_cast<byte>(static_cast<unsigned char>(
      static_cast<unsigned int>(b) >> shift));
```

```cpp
constexpr byte& operator|=(byte& l, byte r) noexcept;
9
   Effects: Equivalent to:
   return l = l | r;
```

```cpp
constexpr byte operator|(byte l, byte r) noexcept;
10
   Effects: Equivalent to:
   return static_cast<byte>(static_cast<unsigned char>(
      static_cast<unsigned int>(l) | 
      static_cast<unsigned int>(r)));
```

\(^{190}\) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

§ 21.2.5
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;

Remarks: This function shall not participate in overload resolution unless is_integral_v<IntType>
is true.

Effects: Equivalent to: return static_cast<IntType>(b);

21.3 Implementation properties  [support.limits]
21.3.1 General  [support.limits.general]

The headers <limits> (21.3.2), <climits> (21.3.5), and <cfloat> (21.3.6) supply characteristics of imple-
mementation-dependent arithmetic types (6.7.1).

21.3.2 Header <limits> synopsis  [limits.syn]

namespace std {

// 21.3.3, floating-point type properties
enum float_round_style;
enum float_denorm_style;

// 21.3.4, 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<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>;

§ 21.3.2 438
template<> class numeric_limits<float>;
template<> class numeric_limits<double>;
template<> class numeric_limits<long double>;
}

21.3.3 Floating-point type properties

21.3.3.1 Type float_round_style

namespace std {
  enum float_round_style {
    round_indeterminate = -1,
    round_toward_zero = 0,
    round_to_nearest = 1,
    round_toward_infinity = 2,
    round_toward_neg_infinity = 3
  };
}

1 The rounding mode for floating-point arithmetic is characterized by the values:

(1.1) — round_indeterminate if the rounding style is indeterminable
(1.2) — round_toward_zero if the rounding style is toward zero
(1.3) — round_to_nearest if the rounding style is to the nearest representable value
(1.4) — round_toward_infinity if the rounding style is toward infinity
(1.5) — round_toward_neg_infinity if the rounding style is toward negative infinity

21.3.3.2 Type float_denorm_style

namespace std {
  enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
  };
}

1 The presence or absence of subnormal numbers (variable number of exponent bits) is characterized by the values:

(1.1) — denorm_indeterminate if it cannot be determined whether or not the type allows subnormal values
(1.2) — denorm_absent if the type does not allow subnormal values
(1.3) — denorm_present if the type does allow subnormal values

21.3.4 Class template numeric_limits

1 The numeric_limits class template provides a C++ program with information about various properties of the implementation’s representation of the arithmetic types.

namespace std {
  template<class T> class numeric_limits {
    public:
      static constexpr bool is_specialized = false;
      static constexpr T min() noexcept { return T(); }
      static constexpr T max() noexcept { return T(); }
      static constexpr T lowest() noexcept { return T(); }
      static constexpr int digits = 0;
      static constexpr int digits10 = 0;
      static constexpr int max_digits10 = 0;
      static constexpr bool is_signed = false;
      static constexpr bool is_integer = false;
      static constexpr bool is_exact = false;
      static constexpr int radix = 0;
      static constexpr T epsilon() noexcept { return T(); }
      static constexpr T round_error() noexcept { return T(); }
  }
}
static constexpr int min_exponent = 0;
static constexpr int min_exponent10 = 0;
static constexpr int max_exponent = 0;
static constexpr int max_exponent10 = 0;

static constexpr bool has_infinity = false;
static constexpr bool has_quiet_NaN = false;
static constexpr bool has_signaling_NaN = false;
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;
static constexpr T infinity() noexcept { return T(); }
static constexpr T quiet_NaN() noexcept { return T(); }
static constexpr T signaling_NaN() noexcept { return T(); }
static constexpr T denorm_min() noexcept { return T(); }

static constexpr bool is_iec559 = false;
static constexpr bool is_bounded = false;
static constexpr bool is_modulo = false;
static constexpr bool traps = false;
static constexpr bool tinyness_before = false;
static constexpr float_round_style round_style = round_toward_zero;

2 For all members declared static constexpr in the numeric_limits template, specializations shall define these values in such a way that they are usable as constant expressions.

3 The default numeric_limits<T> template shall have all members, but with 0 or false values.

4 Specializations shall be provided for each arithmetic type, both floating-point and integer, including bool. The member is_specialized shall be true for all such specializations of numeric_limits.

5 The value of each member of a specialization of numeric_limits on a cv-qualified type cv T shall be equal to the value of the corresponding member of the specialization on the unqualified type T.

6 Non-arithmetic standard types, such as complex<T> (29.5.2), shall not have specializations.

21.3.4.1 numeric_limits members

1 Each member function defined in this subclause is signal-safe (21.11.4).

static constexpr T min() noexcept;
2 Minimum finite value.\(^\text{191}\)
3 For floating types with subnormal numbers, returns the minimum positive normalized value.
4 Meaningful for all specializations in which is_bounded != false, or is_bounded == false && is_-signed == false.

static constexpr T max() noexcept;
5 Maximum finite value.\(^\text{192}\)
6 Meaningful for all specializations in which is_bounded != false.

static constexpr T lowest() noexcept;
7 A finite value x such that there is no other finite value y where y < x.\(^\text{193}\)
8 Meaningful for all specializations in which is_bounded != false.

---

\(^\text{191}\) Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc.
\(^\text{192}\) Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.
\(^\text{193}\) 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.
static constexpr int digits;
  Number of \texttt{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 \texttt{radix} digits in the mantissa.\footnote{Equivalent to \texttt{FLT\_MANT\_DIG}, \texttt{DBL\_MANT\_DIG}, \texttt{LDBL\_MANT\_DIG}.}

static constexpr int digits10;
  Number of base 10 digits that can be represented without change.\footnote{Equivalent to \texttt{FLT\_DIG}, \texttt{DBL\_DIG}, \texttt{LDBL\_DIG}.}
  Meaningful for all specializations in which \texttt{is\_bounded} \neq \texttt{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;
  \texttt{true} if the type is signed.
  Meaningful for all specializations.

static constexpr bool is_integer;
  \texttt{true} if the type is integer.
  Meaningful for all specializations.

static constexpr bool is_exact;
  \texttt{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 types, specifies the base or \texttt{radix} of the exponent representation (often 2).\footnote{Equivalent to \texttt{FLT\_RADIX}.}
  For integer types, specifies the base of the representation.\footnote{Distinguishes types with bases other than 2 (e.g. BCD).}
  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.\footnote{Equivalent to \texttt{FLT\_EPSILON}, \texttt{DBL\_EPSILON}, \texttt{LDBL\_EPSILON}.}
  Meaningful for all floating-point types.

static constexpr T round_error() noexcept;
  Measure of the maximum rounding error.\footnote{Rounding error is described in LIA-1 Section 5.2.4 and Annex C Rationale Section C.5.2.4 — Rounding and rounding constants.}

static constexpr int min_exponent;
  Minimum negative integer such that \texttt{radix} raised to the power of one less than that integer is a normalized floating-point number.\footnote{Equivalent to \texttt{FLT\_MIN\_EXP}, \texttt{DBL\_MIN\_EXP}, \texttt{LDBL\_MIN\_EXP}.}
  Meaningful for all floating-point types.

static constexpr int min_exponent10;
  Minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.\footnote{Equivalent to \texttt{FLT\_MIN\_10\_EXP}, \texttt{DBL\_MIN\_10\_EXP}, \texttt{LDBL\_MIN\_10\_EXP}.}
Meaningful for all floating-point types.

```cpp
static constexpr int max_exponent;
```

Maximal positive integer such that `radix` raised to the power one less than that integer is a representable finite floating-point number.\(^{202}\)

Meaningful for all floating-point types.

```cpp
static constexpr int max_exponent10;
```

Maximal positive integer such that 10 raised to that power is in the range of representable finite floating-point numbers.\(^{203}\)

Meaningful for all floating-point types.

```cpp
static constexpr bool has_infinity;
```

true if the type has a representation for positive infinity.\(^{204}\)

Meaningful for all floating-point types.

```cpp
static constexpr bool has_quiet_NaN;
```

true if the type has a representation for a quiet (non-signaling) “Not a Number”.\(^{205}\)

Meaningful for all floating-point types.

```cpp
static constexpr bool has_signaling_NaN;
```

true if the type has a representation for a signaling “Not a Number”.\(^{205}\)

Meaningful for all floating-point types.

```cpp
static constexpr float_denorm_style has_denorm;
```

denorm_present if the type allows subnormal values (variable number of exponent bits)\(^{206}\),
denorm_absent if the type does not allow subnormal values, and
denorm_indeterminate if it is indeterminate at compile time whether the type allows subnormal values.

Meaningful for all floating-point types.

```cpp
static constexpr bool has_denorm_loss;
```

true if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.\(^{207}\)

```cpp
static constexpr T infinity() noexcept;
```

Representation of positive infinity, if available.\(^{208}\)

Meaningful for all specializations for which `has_infinity` ! = false. Required in specializations for which `is_iec559` ! = false.

```cpp
static constexpr T quiet_NaN() noexcept;
```

Representation of a quiet “Not a Number”, if available.\(^{209}\)

Meaningful for all specializations for which `has_quiet_NaN` ! = false. Required in specializations for which `is_iec559` ! = false.

\(^{202}\) Equivalent to `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.

\(^{203}\) Equivalent to `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.

\(^{204}\) Required by LIA-1.

\(^{205}\) Required by LIA-1.

\(^{206}\) Required by LIA-1.

\(^{207}\) Required by LIA-1.

\(^{208}\) Required by LIA-1.

\(^{209}\) Required by LIA-1.
static constexpr T signaling_NaN() noexcept;
52 Representation of a signaling “Not a Number”, if available.\(^{210}\)
53 Meaningful for all specializations for which \texttt{has\_signaling\_NaN} != false. Required in specializations
54 for which \texttt{is\_iec559} != false.

static constexpr T denorm_min() noexcept;
54 Minimum positive subnormal value.\(^{211}\)
55 Meaningful for all floating-point types.
56 In specializations for which \texttt{has\_denorm} == false, returns the minimum positive normalized value.

static constexpr bool is_iec559;
56 true if and only if the type adheres to ISO/IEC/IEEE 60559.\(^{212}\)
57 Meaningful for all floating-point types.

static constexpr bool is_bounded;
56 true if the set of values representable by the type is finite.\(^{213}\) [Note: All fundamental types (6.7.1) are
57 bounded. This member would be false for arbitrary precision types. — end note]
58 Meaningful for all specializations.

static constexpr bool is_modulo;
61 true if the type is modulo.\(^{214}\) A type is modulo if, for any operation involving +, -, or * on values of
62 that type whose result would fall outside the range \([\min(), \max()]\), the value returned differs from
63 the true value by an integer multiple of \(\max() - \min() + 1\).
64 [Example: \texttt{is\_modulo} is false for signed integer types (6.7.1) unless an implementation, as an extension
65 to this document, defines signed integer overflow to wrap. — end example]
66 Meaningful for all specializations.

static constexpr bool traps;
64 true if, at program startup, there exists a value of the type that would cause an arithmetic operation
65 using that value to trap.\(^{215}\)
66 Meaningful for all specializations.

static constexpr bool tinyness_before;
66 true if tinyness is detected before rounding.\(^{216}\)
67 Meaningful for all floating-point types.

static constexpr float_round_style round_style;
68 The rounding style for the type.\(^{217}\)
69 Meaningful for all floating-point types. Specializations for integer types shall return \texttt{round\_toward\_-zero}.

\[ 21.3.4.2 \text{ numeric\_limits} \text{ specializations} \] [numeric\_special]

1 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 false). Any value
that is not “meaningful” shall be set to 0 or false.

2 [Example:]

\(^{210}\) Required by LIA-1.
\(^{211}\) Required by LIA-1.
\(^{213}\) Required by LIA-1.
\(^{214}\) Required by LIA-1.
\(^{215}\) Equivalent to \texttt{FLT\_ROUNDS}. Required by LIA-1.
\(^{216}\) Refer to ISO/IEC/IEEE 60559. Required by LIA-1.
\(^{217}\) Equivalent to \texttt{FLT\_ROUNDS}. Required by LIA-1.

\[ \]$ 21.3.4.2$ 443
namespace std {
    template<> class numeric_limits<float> {
    public:
        static constexpr bool is_specialized = true;

        static constexpr float min() noexcept { return 1.17549435E-38F; }
        static constexpr float max() noexcept { return 3.40282347E+38F; }
        static constexpr float lowest() noexcept { return -3.40282347E+38F; }
        static constexpr int digits = 24;
        static constexpr int digits10 = 6;
        static constexpr int max_digits10 = 9;

        static constexpr bool is_signed = true;
        static constexpr bool is_integer = false;
        static constexpr bool is_exact = false;

        static constexpr int radix = 2;
        static constexpr float epsilon() noexcept { return 1.19209290E-07F; }
        static constexpr float round_error() noexcept { return 0.5F; }
        static constexpr int min_exponent = -125;
        static constexpr int min_exponent10 = -37;
        static constexpr int max_exponent = +128;
        static constexpr int max_exponent10 = +38;

        static constexpr bool has_infinity = true;
        static constexpr bool has_quiet_NaN = true;
        static constexpr bool has_signaling_NaN = true;
        static constexpr float_denorm_style has_denorm = denorm_absent;
        static constexpr bool has_denorm_loss = false;

        static constexpr float infinity() noexcept { return value; }
        static constexpr float quiet_NaN() noexcept { return value; }
        static constexpr float signaling_NaN() noexcept { return value; }
        static constexpr float denorm_min() noexcept { return min(); }

        static constexpr bool is_iec559 = true;
        static constexpr bool is_bounded = true;
        static constexpr bool is_modulo = false;
        static constexpr bool traps = true;
        static constexpr bool tinyness_before = true;

        static constexpr float_round_style round_style = round_to_nearest;
    };
}
— end example]

3 The specialization for 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;

§ 21.3.4.2
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 int max_exponent = 0;
static constexpr int max_exponent10 = 0;
static constexpr bool 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;

21.3.5 Header <climits> synopsis
[climits.syn]
#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 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

21.3.6 Header <cfloat> synopsis
[cfloat.syn]
#define FLT_ROUNDS see below
#define FLT_EVAL_METHOD see below
#define FLT_HAS_SUBNORM see below
#define DBL_HAS_SUBNORM see below
#define LDBL_HAS_SUBNORM see below
#define FLT_RADIX see below
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

21.4 Integer types

21.4.1 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;
  using int_least8_t = signed integer type;
  using int_least16_t = signed integer type;
  using int_least32_t = signed integer type;
  using int_least64_t = signed integer type;
  using intmax_t = signed integer type;
  using intptr_t = signed integer type; // optional
  using uint8_t = unsigned integer type; // optional
  using uint16_t = unsigned integer type; // optional
  using uint32_t = unsigned integer type; // optional
  using uint64_t = unsigned integer type; // optional
}
```
using uint_fast8_t  = unsigned integer type;
using uint_fast16_t = unsigned integer type;
using uint_fast32_t = unsigned integer type;
using uint_fast64_t = unsigned integer type;
using uint_least8_t = unsigned integer type;
using uint_least16_t = unsigned integer type;
using uint_least32_t = unsigned integer type;
using uint_least64_t = unsigned integer type;
using uintmax_t     = unsigned integer type;
using uintptr_t     = unsigned integer type; // optional
}

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>.

See also: ISO C 7.20

21.5 Start and termination

[support.start.term]

1 [ Note: The header <cstdlib> (21.2.2) declares the functions described in this subclause. — end note ]

[[noreturn]] void _Exit(int status) noexcept;

2 Effects: This function has the semantics specified in the C standard library.

3 Remarks: The program is terminated without executing destructors for objects of automatic, thread,
or static storage duration and without calling functions passed to atexit() (6.8.3.4). The function
_Exit is signal-safe (21.11.4).

[[noreturn]] void abort() noexcept;

4 Effects: This function has the semantics specified in the C standard library.

5 Remarks: The program is terminated without executing destructors for objects of automatic, thread,
or static storage duration and without calling functions passed to atexit() (6.8.3.4). The function
abort is signal-safe (21.11.4).

int atexit(c-atexit-handler* f) noexcept;
int atexit(atexit-handler* f) noexcept;

6 Effects: The atexit() functions register the function pointed to by f to be called without arguments
at normal program termination. It is unspecified whether a call to atexit() that does not happen
before (6.8.2) a call to exit() will succeed. [ Note: The atexit() functions do not introduce a data
race (20.5.5.9). — end note ]

7 Implementation limits: The implementation shall support the registration of at least 32 functions.

8 Returns: The atexit() function returns zero if the registration succeeds, nonzero if it fails.

[[noreturn]] void exit(int status);

9 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

§ 21.5
are called. See 6.8.3.4 for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling \texttt{exit}.)

If control leaves a registered function called by \texttt{exit} because the function does not provide a handler for a thrown exception, \texttt{std::terminate} shall be called (18.5.1).

— Next, all open C streams (as mediated by the function signatures declared in \texttt{<cstdio>}) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling \texttt{tmpfile} are removed.

— Finally, control is returned to the host environment. If \texttt{status} is zero or \texttt{EXIT\_SUCCESS}, an implementation-defined form of the status \texttt{successful\_termination} is returned. If \texttt{status} is \texttt{EXIT\_FAILURE}, an implementation-defined form of the status \texttt{unsuccessful\_termination} is returned. Otherwise the status returned is implementation-defined.

\begin{verbatim}
int at_quick_exit(c-atexit-handler* f) noexcept;
int at_quick_exit(atexit-handler* f) noexcept;
\end{verbatim}

10 Effects: The \texttt{at_quick_exit}() functions register the function pointed to by \texttt{f} to be called without arguments when \texttt{quick_exit} is called. It is unspecified whether a call to \texttt{at_quick_exit}() that does not happen before (6.8.2) all calls to \texttt{quick_exit} will succeed. [Note: The \texttt{at_quick_exit}() functions do not introduce a data race (20.5.5.9). — end note] [Note: The order of registration may be indeterminate if \texttt{at_quick_exit} was called from more than one thread. — end note] [Note: The \texttt{at_quick_exit} registrations are distinct from the \texttt{atexit} registrations, and applications may need to call both registration functions with the same argument. — end note]

11 Implementation limits: The implementation shall support the registration of at least 32 functions.

12 Returns: Zero if the registration succeeds, nonzero if it fails.

\begin{verbatim}
[[noreturn]] void quick_exit(int status) noexcept;
\end{verbatim}

13 Effects: Functions registered by calls to \texttt{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 \texttt{quick_exit}. If control leaves a registered function called by \texttt{quick_exit} because the function does not provide a handler for a thrown exception, \texttt{std::terminate}() shall be called. [Note: A function registered via \texttt{at_quick_exit} is invoked by the thread that calls \texttt{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, \texttt{quick_exit} shall call \texttt{_Exit(status)}.

14 Remarks: The function \texttt{quick_exit} is signal-safe (21.11.4) when the functions registered with \texttt{at\_quick_exit} are.

See also: ISO C 7.22.4

21.6 Dynamic memory management

The header \texttt{<new>} defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

21.6.1 Header \texttt{<new>} synopsis

\begin{verbatim}
namespace std {
    class bad_alloc;
    class bad_array_new_length;

    enum class align_val_t : size_t {};

    struct nothrow_t { explicit nothrow_t() = default; }; external const nothrow_t nothrow;
\end{verbatim}
using new_handler = void (*)(void*);
new_handler get_new_handler() noexcept;
new_handler set_new_handler(new_handler new_p) noexcept;

// 21.6.4, pointer optimization barrier
template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;

// 21.6.5, hardware interference size
inline constexpr size_t hardware_destructive_interference_size = implementation-defined;
inline constexpr size_t hardware_constructive_interference_size = implementation-defined;

[[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;

21.6.2 Storage allocation and deallocation

Except where otherwise specified, the provisions of 6.6.4.4 apply to the library versions of operator new and operator delete. If the value of an alignment argument passed to any of these functions is not a valid alignment value, the behavior is undefined.

21.6.2.1 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 (6.6.4.4.1) called by a new-expression (8.5.2.4) 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 (6.6.4.4), or else throw a bad_alloc exception. This requirement is binding on any replacement versions of these functions.

Default behavior:
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.

Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the current new_handler (21.6.3.5) is a null pointer value, throws bad_alloc.

Otherwise, the function calls the current new_handler function (21.6.3.3). If the called function returns, the loop repeats.

The loop terminates when an attempt to allocate the requested storage is successful or when a called new_handler function does not return.

```cpp
[[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 (6.6.4.4), 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.

```cpp
[[end example]]
```

```cpp
T* p1 = new T;
  // throws bad_alloc if it fails
T* p2 = new(nothrow) T;
  // returns nullptr if it fails
```

**Example:**

**Effects:** The deallocation functions (6.6.4.4.2) called by a delete-expression (8.5.2.5) 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]

**Requires:** ptr shall be a null pointer or its value shall represent 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.

**Requires:** If an implementation has strict pointer safety (6.6.4.4.3) then ptr shall be a safely-derived pointer.

**Requires:** If the alignment parameter is not present, ptr shall have been returned by an allocation function without an alignment parameter. If present, the alignment argument shall equal the alignment argument passed to the allocation function that returned ptr. If present, the size argument shall equal 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]

§ 21.6.2.1
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>.

```cpp
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 (6.6.4.4.2) called by the implementation to render the value of ptr invalid when the constructor invoked from a nothrow placement version of the new-expression throws an exception.

Replaceable: A C++ program may define functions with either of these function signatures, and thereby displace the default versions defined by the C++ standard library.

Requires: ptr shall be a null pointer or its value shall represent 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.

Requires: If an implementation has strict pointer safety (6.6.4.4.3) then ptr shall be a safely-derived pointer.

Requires: If the alignment parameter is not present, ptr shall have been returned by an allocation function without an alignment parameter. If present, the alignment argument shall equal the alignment argument passed to the allocation function that returned ptr.

Default behavior: Calls operator delete(ptr), or operator delete(ptr, alignment), respectively.

### 21.6.2.2 Array forms

```cpp
[[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 (6.6.4.4.1) called by the array form of a new-expression (8.5.2.4) 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.\(^{221}\)

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.

```cpp
[[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 (6.6.4.4), 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.

\(^{221}\) 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.
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.

```cpp
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 (6.6.4.4.2) 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]

**Requires:** `ptr` shall be a null pointer or its value shall represent 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[]`.

**Requires:** If an implementation has strict pointer safety (6.6.4.4.3) then `ptr` shall be a safely-derived pointer.

**Requires:** If the `alignment` parameter is not present, `ptr` shall have been returned by an allocation function without an `alignment` parameter. If present, the `alignment` argument shall equal the `alignment` argument passed to the allocation function that returned `ptr`. If present, the `size` argument shall equal 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.

```cpp
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 (6.6.4.4.2) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a noexcept 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.

**Requires:** `ptr` shall be a null pointer or its value shall represent 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[]`.

**Requires:** If an implementation has strict pointer safety (6.6.4.4.3) then `ptr` shall be a safely-derived pointer.

**Requires:** If the `alignment` parameter is not present, `ptr` shall have been returned by an allocation function without an `alignment` parameter. If present, the `alignment` argument shall equal the `alignment` argument passed to the allocation function that returned `ptr`.

**Default behavior:** Calls `operator delete[](ptr)`, or `operator delete[](ptr, alignment)`, respectively.

### 21.6.2.3 Non-allocating forms

These functions are reserved; a C++ program may not define functions that displace the versions in the C++ standard library (20.5.4). The provisions of 6.6.4.4 do not apply to these reserved placement forms of

---

§ 21.6.2.3 452
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.

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

Returns: ptr.

Remarks: Intentionally performs no other action.

void operator delete(void* ptr, void*) noexcept;

Effects: Intentionally performs no action.

Requires: If an implementation has strict pointer safety (6.6.4.4.3) then ptr shall be a safely-derived pointer.

Remarks: Default function called when any part of the initialization in a placement new-expression that invokes the library’s non-array placement operator new terminates by throwing an exception (8.5.2.4).

void operator delete[](void* ptr, void*) noexcept;

Effects: Intentionally performs no action.

Requires: If an implementation has strict pointer safety (6.6.4.4.3) then ptr shall be a safely-derived pointer.

Remarks: Default function called when any part of the initialization in a placement new-expression that invokes the library’s array placement operator new terminates by throwing an exception (8.5.2.4).

21.6.2.4 Data races

For purposes of determining the existence of data races, the library versions of operator new, user replacement versions of global operator new, the C standard library functions `aligned_alloc`, `calloc`, and `malloc`, the library versions of operator delete, user replacement versions of operator delete, the C standard library function `free`, and the C standard library function `realloc` shall not introduce a data race (20.5.5.9). Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before (6.8.2) the next allocation (if any) in this order.

21.6.3 Storage allocation errors

21.6.3.1 Class bad_alloc

```cpp
namespace std {
    class bad_alloc : public exception {
    public:
        bad_alloc() noexcept;
        bad_alloc(const bad_alloc&) noexcept;
        bad_alloc& operator=(const bad_alloc&) noexcept;
        const char* what() const noexcept override;
    }
}
```

The class bad_alloc defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

```cpp
bad_alloc() noexcept;
```

Effects: Constructs an object of class bad_alloc.

```cpp
bad_alloc(const bad_alloc&) noexcept;
bad_alloc& operator=(const bad_alloc&) noexcept;
```

Effects: Copies an object of class bad_alloc.
const char* what() const noexcept override;

Returns: An implementation-defined ntbs.

Remarks: The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a wstring (24.3, 25.4.1.4).

21.6.3.2 Class bad_array_new_length

namespace std {
    class bad_array_new_length : public bad_alloc {
    public:
        bad_array_new_length() noexcept;
        const char* what() const noexcept override;
    }
}

1 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 (8.5.2.4).

bad_array_new_length() noexcept;

Effects: Constructs an object of class bad_array_new_length.

const char* what() const noexcept override;

Returns: An implementation-defined ntbs.

Remarks: The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a wstring (24.3, 25.4.1.4).

21.6.3.3 Type new_handler

using new_handler = void (*)();

1 The type of a handler function to be called by operator new() or operator new[] () (21.6.2) when they cannot satisfy a request for additional storage.

Required behavior: A new_handler shall perform one of the following:

(2.1) — make more storage available for allocation and then return;
(2.2) — throw an exception of type bad_alloc or a class derived from bad_alloc;
(2.3) — terminate execution of the program without returning to the caller.

21.6.3.4 set_new_handler

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.

21.6.3.5 get_new_handler

new_handler get_new_handler() noexcept;

Returns: The current new_handler. [Note: This may be a null pointer value. — end note]

21.6.4 Pointer optimization barrier

template<class T> [[nodiscard]] constexpr T* launder(T* p) noexcept;

Requires: 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 (7.5) to T is located at the address A. All bytes of storage that would be reachable through the result are reachable through p (see below).

Returns: A value of type T * that points to X.

Remarks: An invocation of this function may be used in a core constant expression whenever the value of its argument may be used in a core constant expression. A byte of storage 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. The program is ill-formed if $T$ is a function type or $cv$ void.

[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 $\text{const}$ 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]

21.6.5 Hardware interference size

```cpp
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 \text{alignof}(\text{max align t}).

[Example:
```cpp
struct keep_apart {
    alignas(hardware_destructive_interference_size) atomic<int> cat;
    alignas(hardware_destructive_interference_size) atomic<int> dog;
};
```
—end example]

```cpp
inline constexpr size_t hardware_constructive_interference_size = implementation-defined;
```

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 \text{alignof}(\text{max align t}).

[Example:
```cpp
struct together {
    atomic<int> dog;
    int puppy;
};
```
```cpp
struct kennel {
    // Other data members...
    alignas(sizeof(together)) together pack;
    // Other data members...
};
```
```cpp
static_assert(sizeof(together) <= hardware_constructive_interference_size);
```
—end example]

21.7 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.

21.7.1 Header `<typeinfo>` synopsis

```cpp
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
}
```
21.7.2 Class type_info

```cpp
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
    };
}
```

The class `type_info` describes type information generated by the implementation (8.5.1.8). Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

```cpp
bool operator==(const type_info& rhs) const noexcept;
```

**Effects:** Compares the current object with `rhs`.

**Returns:** `true` if the two values describe the same type.

```cpp
bool operator!=(const type_info& rhs) const noexcept;
```

**Returns:** `!(*this == rhs)`.

```cpp
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.

```cpp
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.

**Remarks:** An implementation should return different values for two `type_info` objects which do not compare equal.

```cpp
const char* name() const noexcept;
```

**Returns:** An implementation-defined `ntbs`.

**Remarks:** The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a `wstring` (24.3, 25.4.1.4)

21.7.3 Class bad_cast

```cpp
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;
    };
}
```

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 (8.5.1.7).

```cpp
bad_cast() noexcept;
```

**Effects:** Constructs an object of class `bad_cast`. 

§ 21.7.3 456
bad_cast(const bad_cast&) noexcept;
bad_cast& operator=(const bad_cast&) noexcept;

Effects: Copies an object of class bad_cast.

const char* what() const noexcept override;

Returns: An implementation-defined ntbs.

Remarks: The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a wstring (24.3, 25.4.1.4)

21.7.4 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 (8.5.1.8).

bad_typeid() noexcept;

Effects: Constructs an object of class bad_typeid.

bad_typeid(const bad_typeid&) noexcept;
bad_typeid& operator=(const bad_typeid&) noexcept;

Effects: Copies an object of class bad_typeid.

const char* what() const noexcept override;

Returns: An implementation-defined ntbs.

Remarks: The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a wstring (24.3, 25.4.1.4)

21.8 Exception handling

The header <exception> defines several types and functions related to the handling of exceptions in a C++ program.

21.8.1 Header <exception> synopsis

namespace std {
    class exception;
    class bad_exception;
    class nested_exception;

    using terminate_handler = void (*)();
    terminate_handler get_terminate() noexcept;
    terminate_handler set_terminate(terminate_handler f) noexcept;
    [[noreturn]] void terminate() noexcept;

    int uncaught_exceptions() noexcept;

    using exception_ptr = unspecified;

    exception_ptr current_exception() noexcept;
    [[noreturn]] void rethrow_exception(exception_ptr p);
    template<class E> exception_ptr make_exception_ptr(E e) noexcept;
}
template<class T> [[noreturn]] void throw_with_nested(T&& t);
template<class E> void rethrow_if_nested(const E& e);

21.8.2 Class exception

namespace std {
    class exception {
        public:
            exception() noexcept;
            exception(const exception&) noexcept;
            exception& operator=(const exception&) noexcept;
            virtual ~exception();
            virtual const char* what() const noexcept;
        }
    }

1 The class exception defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

2 Each standard library class T that derives from class exception 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() noexcept;

3 Effects: Constructs an object of class exception.

exception(const exception& rhs) noexcept;

4 Effects: Copies an exception object.

Postconditions: If *this and rhs both have dynamic type exception then the value of the expression strcmp(what(), rhs.what()) shall equal 0.

virtual ~exception();

5 Effects: Destroys an object of class exception.

virtual const char* what() const noexcept override;

6 Returns: An implementation-defined ntbs.

Remarks: The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a wstring (24.3, 25.4.1.4). 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.

21.8.3 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;
        }
    }

1 The class bad_exception defines the type of the object referenced by the exception_ptr returned from a call to current_exception (21.8.6) when the currently active exception object fails to copy.

bad_exception() noexcept;

2 Effects: Constructs an object of class bad_exception.

bad_exception(const bad_exception&) noexcept;

3 Effects: Copies an object of class bad_exception.
const char* what() const noexcept override;

*Returns:* An implementation-defined NTBS.

*Remarks:* The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a *wstring* (24.3, 25.4.1.4).

### 21.8.4 Abnormal termination

#### 21.8.4.1 Type terminate_handler

```cpp
using terminate_handler = 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()`.

#### 21.8.4.2 set_terminate

```cpp
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`.

#### 21.8.4.3 get_terminate

```cpp
terminate_handler get_terminate() noexcept;
```

*Returns:* The current `terminate_handler`. [*Note: This may be a null pointer value. —end note*]

#### 21.8.4.4 terminate

```cpp
[[noreturn]] void terminate() noexcept;
```

*Remarks:* Called by the implementation when exception handling must be abandoned for any of several reasons (18.5.1). 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*]

### 21.8.5 uncaught_exceptions

```cpp
int uncaught_exceptions() noexcept;
```

*Returns:* The number of uncaught exceptions (18.5.2).

*Remarks:* When `uncaught_exceptions() > 0`, throwing an exception can result in a call of `std::terminate()` (18.5.1).

### 21.8.6 Exception propagation

```cpp
using exception_ptr = unspecified;
```

*The type `exception_ptr` can be used to refer to an exception object.*

*exception_ptr shall satisfy the requirements of NullablePointer (20.5.3.3).*

*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.*

*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]

```
7 exception_ptr current_exception() noexcept;
8 Returns: An `exception_ptr` object that refers to the currently handled exception (18.3) 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]
```

```
9 void rethrow_exception(exception_ptr p);
10 Requires: p shall not be a null pointer.
11 Throws: The exception object to which p refers.
```

```
11 template<class E> exception_ptr make_exception_ptr(E e) noexcept;
12 Effects: Creates an `exception_ptr` object that refers to a copy of e, as if:
13 try {
14    throw e;
15 } catch(...) {
16    return current_exception();
17 }
18 [ Note: This function is provided for convenience and efficiency reasons. —end note]
```

### § 21.8.7 `nested_exception` [except.nested]

```
namespace std {
    class nested_exception {
        public:
            nested_exception() noexcept;
            nested_exception(const nested_exception&) noexcept = default;
            nested_exception& operator=(const nested_exception&) noexcept = default;
            virtual ~nested_exception() = default;

            // access functions
            [[noreturn]] void rethrow_nested() const;
            exception_ptr nested_ptr() const noexcept;
        }

    template<class T> [[noreturn]] void throw_with_nested(T& t);
    template<class E> void rethrow_if_nested(const E& e);
}
```

1 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.

2 [Note: `nested_exception` has a virtual destructor to make it a polymorphic class. Its presence can be tested for with `dynamic_cast`. —end note]

```
2 nested_exception() noexcept;
3 Effects: The constructor calls `current_exception()` and stores the returned value.
```
[[noreturn]] void rethrow_nested() const;

**Effects:** If nested_ptr() returns a null pointer, the function calls std::terminate(). Otherwise, it throws the stored exception captured by *this.

exception_ptr nested_ptr() const noexcept;

**Returns:** The stored exception captured by this nested_exception object.

template<class T> [[noreturn]] void throw_with_nested(T&& t);

Let \( U \) be decay_t<T>.

**Requires:** \( U \) shall be CopyConstructible.

**Throws:** If is_class_v<\( U \)>&& !is_final_v<\( U \)>&& !is_base_of_v<nested_exception, \( U \)> is true, an exception of unspecified type that is publicly derived from both \( U \) and nested_exception and constructed from std::forward<\( T \)>(\( t \)), otherwise std::forward<\( T \)>(\( t \)).

template<class E> void rethrow_if_nested(const E& e);

**Effects:** If \( E \) is not a polymorphic class type, or if nested_exception is an inaccessible or ambiguous base class of \( E \), there is no effect. Otherwise, performs:

```cpp
if (auto p = dynamic_cast<const nested_exception*>(addressof(e)))
    p->rethrow_nested();
```

### 21.9 Initializer lists

The header `<initializer_list>` defines a class template and several support functions related to list-initialization (see 11.6.4). All functions specified in this subclause are signal-safe (21.11.4).

#### 21.9.1 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;
            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
    };

    template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;
    template<class E> constexpr const E* end(initializer_list<E> il) noexcept;
}
```

An object of type `initializer_list<E>` provides access to an array of objects of type `const E`. [Note: A pair of pointers or a pointer plus a length would be obvious representations for `initializer_list`. `initializer_list` is used to implement initializer lists as specified in 11.6.4. Copying an initializer list does not copy the underlying elements. — end note]

If an explicit specialization or partial specialization of `initializer_list` is declared, the program is ill-formed.

#### 21.9.2 Initializer list constructors

```cpp
constexpr initializer_list() noexcept;

**Effects:** Constructs an empty `initializer_list` object.
Postconditions: \( \text{size()} = 0 \).

### 21.9.3 Initializer list access

- constexpr const E* begin() const noexcept;

  Returns: A pointer to the beginning of the array. If \( \text{size()} = 0 \) the values of \text{begin()} and \text{end()} are unspecified but they shall be identical.

- constexpr const E* end() const noexcept;

  Returns: \( \text{begin()} + \text{size()} \).

- constexpr size_t size() const noexcept;

  Returns: The number of elements in the array.

  Complexity: Constant time.

### 21.9.4 Initializer list range access

- template<class E> constexpr const E* begin(initializer_list<E> il) noexcept;

  Returns: \( \text{il.begin()} \).

- template<class E> constexpr const E* end(initializer_list<E> il) noexcept;

  Returns: \( \text{il.end()} \).

### 21.10 Comparisons

#### 21.10.1 Header \text{<compare>} synopsis

The header \text{<compare>} specifies types, objects, and functions for use primarily in connection with the three-way comparison operator (8.5.8).

```cpp
namespace std {

  // 21.10.2, 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; }

  // 21.10.3, 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;

  // 21.10.4, 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);

} // namespace std
```

§ 21.10.1
21.10.2 Comparison category types [cmp.categories]

21.10.2.1 Preamble [cmp.categories.pre]

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]

3 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.

4 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.

21.10.2.2 Class weak_equality [cmp.weakeq]

1 The weak_equality type is typically used as the result type of a three-way comparison operator (8.5.8) that (a) admits only equality and inequality comparisons, and (b) does not imply substitutability.

```cpp
namespace std {
    class weak_equality {
    private:
        int value; // exposition only

    // exposition-only constructor
    explicit constexpr 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 bool operator==(weak_equality v, unspecified) noexcept;
    friend constexpr bool operator!=(weak_equality v, unspecified) noexcept;
    friend constexpr bool operator==(unspecified, weak_equality v) noexcept;
    friend constexpr bool 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);
}
```

```cpp
constexpr bool operator==(weak_equality v, unspecified) noexcept;
constexpr bool operator==(unspecified, weak_equality v) noexcept;
```

2 Returns: v.value == 0.

```cpp
constexpr bool operator!=(weak_equality v, unspecified) noexcept;
constexpr bool operator!=(unspecified, weak_equality v) noexcept;
```

3 Returns: v.value != 0.

21.10.2.3 Class strong_equality [cmp.strongeq]

1 The strong_equality type is typically used as the result type of a three-way comparison operator (8.5.8) that (a) admits only equality and inequality comparisons, and (b) does imply substitutability.

§ 21.10.2.3
namespace std {
  
  class strong_equality {
      int value;  // exposition only
  
      // exposition-only constructor
      explicit constexpr 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!=(strong_equality v, unspecified) noexcept;
      friend constexpr bool operator==(unspecified, strong_equality v) 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);
  }  

  constexpr operator weak_equality() const noexcept {
      value == 0 ? weak_equality::equivalent : weak_equality::nonequivalent.
  
  constexpr bool operator==(strong_equality v, unspecified) noexcept;
  constexpr bool operator==(unspecified, strong_equality v) noexcept;
      Returns: v.value == 0.
  
  constexpr bool operator!=(strong_equality v, unspecified) noexcept;
  constexpr bool operator!=(unspecified, strong_equality v) noexcept;
      Returns: v.value != 0.

  21.10.2.4 Class partial_ordering  [cmp.partialord]
  1

  The partial_ordering type is typically used as the result type of a three-way comparison operator (8.5.8)
  that (a) admits all of the six two-way comparison operators (8.5.9, 8.5.10), (b) does not imply substitutability,
  and (c) permits two values to be incomparable.²²²

  namespace std {
      class partial_ordering {
          int value;  // exposition only
          bool is_ordered;  // exposition only
  
          // exposition-only constructors
          explicit constexpr partial_ordering(eq v) noexcept : value(int(v)), is_ordered(true) {}  // exposition only
          explicit constexpr partial_ordering(ord v) noexcept : value(int(v)), is_ordered(true) {}  // exposition only
          explicit constexpr partial_ordering(ncmp v) noexcept : value(int(v)), is_ordered(false) {}  // exposition only

  ²²² That is, a < b, a == b, and a > b might all be false.
public:
    // 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;

    friend constexpr bool operator== (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator!= (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator< (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator<= (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator> (unspecified, partial_ordering v) noexcept;
    friend constexpr bool operator>= (unspecified, partial_ordering v) noexcept;

    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 && 0 @ v.value.

    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.

21.10.2.5 Class weak_ordering

The weak_ordering type is typically used as the result type of a three-way comparison operator (8.5.8) that (a) admits all of the six two-way comparison operators (8.5.9, 8.5.10), and (b) does not imply substitutability.
// exposition-only constructors
explicit constexpr weak_ordering(eq v) noexcept : value(int(v)) {} // exposition only
explicit constexpr 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;
frend constexpr bool operator!=(weak_ordering v, unspecified) noexcept;
frend constexpr bool operator<(weak_ordering v, unspecified) noexcept;
frend constexpr bool operator<=(weak_ordering v, unspecified) noexcept;
frend constexpr bool operator>(weak_ordering v, unspecified) noexcept;
frend constexpr bool operator>=(weak_ordering v, unspecified) noexcept;
frend constexpr bool operator==(unspecified, weak_ordering v) noexcept;
frend constexpr bool operator!=(unspecified, weak_ordering v) noexcept;
frend constexpr bool operator<(unspecified, weak_ordering v) noexcept;
frend constexpr bool operator<=(unspecified, weak_ordering v) noexcept;
frend constexpr bool operator>(unspecified, weak_ordering v) noexcept;
frend constexpr bool 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@.
21.10.2.6 Class strong_ordering  

The `strong_ordering` type is typically used as the result type of a three-way comparison operator (8.5.8) that (a) admits all of the six two-way comparison operators (8.5.9, 8.5.10), and (b) does imply substitutability.

```cpp
namespace std {
  class strong_ordering {
    int value;  // exposition only
    // exposition-only constructors
    explicit constexpr strong_ordering(eq v) noexcept : value(int(v)) {}  // exposition only
    explicit constexpr strong_ordering(ord v) noexcept : value(int(v)) {}  // exposition only
  public:
    // valid values
    static const strong_ordering less;
    static const strong_ordering equal;
    static const strong_ordering equivalent;
    static const strong_ordering greater;
    // conversions
    constexpr operator 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;
  };
  // 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;

Returns: v.value @ 0 for operator@.

constexpr bool operator==(unspecified, strong_ordering v) noexcept;
constexpr bool operator!=(unspecified, strong_ordering v) noexcept;
constexpr bool operator< (unspecified, strong_ordering v) noexcept;
constexpr bool operator<=(unspecified, strong_ordering v) noexcept;
constexpr bool operator> (unspecified, strong_ordering v) noexcept;
constexpr bool operator>=(unspecified, strong_ordering v) noexcept;

Returns: 0 @ v.value for operator@.

21.10.3 Class template common_comparison_category [cmp.common]

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 (15.9.2) 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]

21.10.4 Comparison algorithms [cmp.alg]

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;
— otherwise, if \( a < b \) is true, returns \texttt{weak\_ordering::less};

— otherwise, returns \texttt{weak\_ordering::greater}.

— Otherwise, the function is defined as deleted.

\[ \text{template<class T> constexpr partial\_ordering partial\_order(const T& a, const T& b);} \]

\[ \text{Effects:}\quad \text{Compares two values and produces a result of type partial\_ordering:} \]

\[ \quad \text{(3.1)} \quad \text{Returns } a \leftrightarrow b \text{ if that expression is well-formed and convertible to partial\_ordering.} \]

\[ \quad \text{(3.2)} \quad \text{Otherwise, if the expression } a \leftrightarrow b \text{ is well-formed, then the function is defined as deleted.} \]

\[ \quad \text{(3.3)} \quad \text{Otherwise, if the expressions } a == b \text{ and } a < b \text{ are each well-formed and convertible to bool, then} \]

\[ \quad \quad \text{(3.3.1)} \quad \text{if } a == b \text{ is true, returns partial\_ordering::equivalent;} \]

\[ \quad \quad \text{(3.3.2)} \quad \text{otherwise, if } a < b \text{ is true, returns partial\_ordering::less;} \]

\[ \quad \quad \text{(3.3.3)} \quad \text{otherwise, returns partial\_ordering::greater.} \]

\[ \quad \text{(3.4)} \quad \text{Otherwise, the function is defined as deleted.} \]

\[ \text{template<class T> constexpr strong\_equality strong\_equal(const T& a, const T& b);} \]

\[ \text{Effects:}\quad \text{Compares two values and produces a result of type strong\_equality:} \]

\[ \quad \text{(4.1)} \quad \text{Returns } a \leftrightarrow b \text{ if that expression is well-formed and convertible to strong\_equality.} \]

\[ \quad \text{(4.2)} \quad \text{Otherwise, if the expression } a \leftrightarrow b \text{ is well-formed, then the function is defined as deleted.} \]

\[ \quad \text{(4.3)} \quad \text{Otherwise, if the expression } a == b \text{ is well-formed and convertible to bool, then} \]

\[ \quad \quad \text{(4.3.1)} \quad \text{if } a == b \text{ is true, returns strong\_equality::equal;} \]

\[ \quad \quad \text{(4.3.2)} \quad \text{otherwise, returns strong\_equality::nonequal.} \]

\[ \quad \text{(4.4)} \quad \text{Otherwise, the function is defined as deleted.} \]

\[ \text{template<class T> constexpr weak\_equality weak\_equal(const T& a, const T& b);} \]

\[ \text{Effects:}\quad \text{Compares two values and produces a result of type weak\_equality:} \]

\[ \quad \text{(5.1)} \quad \text{Returns } a \leftrightarrow b \text{ if that expression is well-formed and convertible to weak\_equality.} \]

\[ \quad \text{(5.2)} \quad \text{Otherwise, if the expression } a \leftrightarrow b \text{ is well-formed, then the function is defined as deleted.} \]

\[ \quad \text{(5.3)} \quad \text{Otherwise, if the expression } a == b \text{ is well-formed and convertible to bool, then} \]

\[ \quad \quad \text{(5.3.1)} \quad \text{if } a == b \text{ is true, returns weak\_equality::equivalent;} \]

\[ \quad \quad \text{(5.3.2)} \quad \text{otherwise, returns weak\_equality::nonequivalent.} \]

\[ \quad \text{(5.4)} \quad \text{Otherwise, the function is defined as deleted.} \]

\[ \text{21.11 Other runtime support} \] [support.runtime]

1 Headers \texttt{<csetjmp>} (nonlocal jumps), \texttt{<csignal>} (signal handling), \texttt{<cstdarg>} (variable arguments), and \texttt{<cstdlib>} (runtime environment \texttt{getenv}, \texttt{system}), provide further compatibility with C code.

2 Calls to the function \texttt{getenv} (21.2.2) shall not introduce a data race (20.5.5.9) provided that nothing modifies the environment. [\textit{Note: Calls to the POSIX functions \texttt{setenv} and \texttt{putenv} modify the environment. —end note}]  

3 A call to the \texttt{setlocale} function (25.5) may introduce a data race with other calls to the \texttt{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 \texttt{locale::global} calls the \texttt{setlocale} function.

\[ \text{21.11.1 Header \texttt{<cstdarg>} synopsis} \] [cstdarg.syn]

\[ \text{namespace std} \{ \]
\[ \quad \text{using va\_list = see below;} \]
\[ \} \]

\[ \S 21.11 \] 469
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 (17.6.3) or an entity resulting from a lambda capture (8.4.5), 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

21.11.2 Header `<csetjmp>` synopsis

```cpp
namespace std {
    using jmp_buf = see below;
    [[noreturn]] void longjmp(jmp_buf env, int val);
}

#define setjmp(env) see below
```

1 The contents of the header `<csetjmp>` are the same as the C standard library header `<setjmp.h>`.

2 The function signature `longjmp(jmp_buf jbuf, int val)` has more restricted behavior in this document. A `setjmp/longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by `catch` and `throw` would invoke any non-trivial destructors for any automatic objects.

SEE ALSO: ISO C 7.13

21.11.3 Header `<csignal>` synopsis

```cpp
namespace std {
    using sig_atomic_t = see below;

    // 21.11.4, 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
```

1 The contents of the header `<csignal>` are the same as the C standard library header `<signal.h>`.

21.11.4 Signal handlers

1 A call to the function `signal` synchronizes with any resulting invocation of the signal handler so installed.

2 A plain lock-free atomic operation is an invocation of a function `f` from Clause 32, 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

223) Note that `va_start` is required to work as specified even if unary `operator&` is overloaded for the type of `parmN`. 

§ 21.11.4
(2.4) \( f \) is a non-member function, and for every pointer-to-atomic argument \( A \) passed to \( f \), \texttt{atomic_is_lock_free}(A) yields \texttt{true}.

3 An evaluation is \textit{signal-safe} unless it includes one of the following:

(3.1) a call to any standard library function, except for plain lock-free atomic operations and functions explicitly identified as signal-safe. [Note: This implicitly excludes the use of \texttt{new} and \texttt{delete} expressions that rely on a library-provided memory allocator. — end note]

(3.2) an access to an object with thread storage duration;

(3.3) a \texttt{dynamic_cast} expression;

(3.4) throwing of an exception;

(3.5) control entering a \texttt{try-block} or \texttt{function-try-block};

(3.6) initialization of a variable with static storage duration requiring dynamic initialization (6.8.3.3, 9.7)\textsuperscript{224}; or

(3.7) waiting for the completion of the initialization of a variable with static storage duration (9.7).

A signal handler invocation has undefined behavior if it includes an evaluation that is not signal-safe.

4 The function \texttt{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.

\textbf{See also:} ISO C 7.14

\textsuperscript{224} Such initialization might occur because it is the first odr-use (6.2) of that variable.
22 Diagnostics library

22.1 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 33.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.2</td>
<td>&lt;stdexcept&gt;</td>
</tr>
<tr>
<td>22.3</td>
<td>&lt;cassert&gt;</td>
</tr>
<tr>
<td>22.4</td>
<td>&lt;cerrno&gt;</td>
</tr>
<tr>
<td>22.5</td>
<td>&lt;system_error&gt;</td>
</tr>
</tbody>
</table>

22.2 Exception classes

The C++ standard library provides classes to be used to report certain errors (20.5.5.12) 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. In theory, they are preventable.

By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header <stdexcept> defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related by inheritance.

22.2.1 Header <stdexcept> synopsis

namespace std {
    class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
    class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}

22.2.2 Class logic_error

namespace std {
    class logic_error : public exception {
        public:
            explicit logic_error(const string& what_arg);
            explicit logic_error(const char* what_arg);
    }
}

The class logic_error defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

logic_error(const string& what_arg);

Effects: Constructs an object of class logic_error.

Postconditions: strcmp(what(), what_arg.c_str()) == 0.
logic_error(const char* what_arg);

Effects: Constructs an object of class logic_error.
Postconditions: strcmp(what(), what_arg) == 0.

### 22.2.3 Class domain_error

```cpp
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);

Effects: Constructs an object of class domain_error.
Postconditions: strcmp(what(), what_arg.c_str()) == 0.

domain_error(const char* what_arg);

Effects: Constructs an object of class domain_error.
Postconditions: strcmp(what(), what_arg) == 0.

### 22.2.4 Class invalid_argument

```cpp
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);

Effects: Constructs an object of class invalid_argument.
Postconditions: strcmp(what(), what_arg.c_str()) == 0.

invalid_argument(const char* what_arg);

Effects: Constructs an object of class invalid_argument.
Postconditions: strcmp(what(), what_arg) == 0.

### 22.2.5 Class length_error

```cpp
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);

Effects: Constructs an object of class length_error.
Postconditions: strcmp(what(), what_arg.c_str()) == 0.
length_error(const char* what_arg);
4
   Effects: Constructs an object of class length_error.
5
   Postconditions: strcmp(what(), what_arg) == 0.

22.2.6 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);
        }
    }
1
The class out_of_range defines the type of objects thrown as exceptions to report an argument value not in its expected range.

out_of_range(const string& what_arg);
2
   Effects: Constructs an object of class out_of_range.
3
   Postconditions: strcmp(what(), what_arg.c_str()) == 0.

out_of_range(const char* what_arg);
4
   Effects: Constructs an object of class out_of_range.
5
   Postconditions: strcmp(what(), what_arg) == 0.

22.2.7 Class runtime_error

namespace std {
    class runtime_error : public exception {
        public:
            explicit runtime_error(const string& what_arg);
            explicit runtime_error(const char* what_arg);
        }
    }
1
The class runtime_error defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

runtime_error(const string& what_arg);
2
   Effects: Constructs an object of class runtime_error.
3
   Postconditions: strcmp(what(), what_arg.c_str()) == 0.

runtime_error(const char* what_arg);
4
   Effects: Constructs an object of class runtime_error.
5
   Postconditions: strcmp(what(), what_arg) == 0.

22.2.8 Class range_error

namespace std {
    class range_error : public runtime_error {
        public:
            explicit range_error(const string& what_arg);
            explicit range_error(const char* what_arg);
        }
    }
1
The class range_error defines the type of objects thrown as exceptions to report range errors in internal computations.

range_error(const string& what_arg);
2
   Effects: Constructs an object of class range_error.
3
   Postconditions: strcmp(what(), what_arg.c_str()) == 0.
range_error(const char* what_arg);

Effects: Constructs an object of class range_error.
Postconditions: strcmp(what(), what_arg) == 0.

22.2.9 Class overflow_error

namespace std {
  class overflow_error : public runtime_error {
    public:
      explicit overflow_error(const string& what_arg);
      explicit overflow_error(const char* what_arg);
    }
  }

The class overflow_error defines the type of objects thrown as exceptions to report an arithmetic overflow error.

overflow_error(const string& what_arg);
Effects: Constructs an object of class overflow_error.
Postconditions: strcmp(what(), what_arg.c_str()) == 0.

overflow_error(const char* what_arg);
Effects: Constructs an object of class overflow_error.
Postconditions: strcmp(what(), what_arg) == 0.

22.2.10 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);
Effects: Constructs an object of class underflow_error.
Postconditions: strcmp(what(), what_arg.c_str()) == 0.

underflow_error(const char* what_arg);
Effects: Constructs an object of class underflow_error.
Postconditions: strcmp(what(), what_arg) == 0.

22.3 Assertions

The header <cassert> provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

22.3.1 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

22.3.2 The assert macro

An expression assert(E) is a constant subexpression (20.3.6), if

---

§ 22.3.2
— E contextually converted to bool (Clause 7) is a constant subexpression that evaluates to the value true.

22.4 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.

22.4.1 Header `<cerrno>` synopsis

```c
#define errno see below
#define E2BIG see below
#define EACCES see below
#define EADDRINUSE see below
#define EADDRNOTAVAIL see below
#define EAFNOSUPPORT see below
#define EAGAIN see below
#define EALREADY see below
#define EBADF see below
#define EBADMSG see below
#define EBUSY see below
#define ECANCELED see below
#define ECHILD see below
#define ECONNABORTED see below
#define ECONNREFUSED see below
#define ECONNRESET see below
#define EDEADLK see below
#define EDESTADDRREQ see below
#define EDOM see below
#define EEXIST see below
#define EFAULT see below
#define EFBIG see below
#define EHOSTUNREACH see below
#define EIDRM see below
#define EILSEQ see below
#define EINPROGRESS see below
#define EINTR see below
#define EINVAL see below
#define EIO see below
#define EISCONN see below
#define EISDIR see below
#define ELOOP see below
#define EMFILE see below
#define EMLINK see below
#define EMSGSIZE see below
#define ENAMETOOLONG see below
#define ENETDOWN see below
#define ENETRESET see below
#define ENETUNREACH see below
#define ENFILE see below
#define ENOBUFS see below
#define ENODATA see below
#define ENODEV see below
#define ENOENT see below
#define ENOEXEC see below
#define ENOLCK see below
#define ENOLINK see below
#define ENOMEM see below
#define ENOMSG see below
#define ENOPROTOOPT see below
#define ENOSPC see below
#define ENOSR see below
#define ENOSTR see below
```
The meaning of the macros in this header is defined by the POSIX standard.

See also: ISO C 7.5

22.5 System error support

1 This subclause describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

2 Components described in this subclause shall not change the value of errno (22.4). Implementations should leave the error states provided by other libraries unchanged.

22.5.1 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 {};

  enum class errc {
    address_family_not_supported, // EAFNOSUPPORT
    address_in_use, // EADDRINUSE
    address_not_available, // EADDRNOTAVAIL
    already_connected, // EISCONN
    argument_list_too_long, // E2BIG
    argument_out_of_domain, // EDMON
    bad_address, // EFAULT
    bad_file_descriptor, // EBADF
    bad_message, // EBADMSG
  }
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 {};

// 22.5.3.5, 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);

// 22.5.4.5, non-member functions
error_condition make_error_condition(errc e) noexcept;

// 22.5.5, comparison functions
bool operator<(const error_code& 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_condition& lhs, const error_condition& rhs) noexcept;
bool operator!=(const error_code& lhs, const error_code& rhs) noexcept;
bool operator!=(const error_condition& lhs, const error_condition& rhs) noexcept;

// 22.5.6, hash support
template<class T> struct hash;
template<> struct hash<error_code>;
template<> struct hash<error_condition>;

// 22.5, system error support
template<class T>
inline constexpr bool is_error_code_enum_v = is_error_code_enum<T>::value;
template<class T>
inline constexpr bool is_error_condition_enum_v = is_error_condition_enum<T>::value;

1 The value of each enum errc constant shall be the same as the value of the <cerrno> macro shown in the above synopsis. Whether or not the <system_error> implementation exposes the <cerrno> macros is unspecified.

2 The is_error_code_enum and is_error_condition_enum may be specialized for user-defined types to indicate that such types are eligible for class error_code and class error_condition automatic conversions, respectively.

22.5.2 Class error_category [syserr.errcat]
22.5.2.1 Class error_category overview [syserr.errcat.overview]
1 The class error_category serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from error_category to support categories of errors in addition to those defined in this document. Such classes shall behave as specified in this subclause 22.5.2. [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;

§ 22.5.2.1 479
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;

22.5.2.2 Class error_category virtual members
virtual ~error_category();
1 Effects: Destroys an object of class error_category.
virtual const char* name() const noexcept = 0;
2 Returns: A string naming the error category.
virtual error_condition default_error_condition(int ev) const noexcept;
3 Returns: error_condition(ev, *this).
virtual bool equivalent(int code, const error_condition& condition) const noexcept;
4 Returns: default_error_condition(code) == condition.
virtual bool equivalent(const error_code& code, int condition) const noexcept;
5 Returns: *this == code.category() && code.value() == condition.
virtual string message(int ev) const = 0;
6 Returns: A string that describes the error condition denoted by ev.

22.5.2.3 Class error_category non-virtual members
constexpr error_category() noexcept;
1 Effects: Constructs an object of class error_category.
bool operator==(const error_category& rhs) const noexcept;
2 Returns: this == &rhs.
bool operator!=(const error_category& rhs) const noexcept;
3 Returns: !(this == rhs).
bool operator<(const error_category& rhs) const noexcept;
4 Returns: less<const error_category*>(this, &rhs).
[Note: less (23.14.7) provides a total ordering for pointers. — end note]

22.5.2.4 Program defined classes derived from error_category
virtual const char* name() const noexcept = 0;
1 Returns: A string naming the error category.
virtual error_condition default_error_condition(int ev) const noexcept;
2 Returns: An object of type error_condition that corresponds to ev.
virtual bool equivalent(int code, const error_condition& condition) const noexcept;

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.

22.5.2.5 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]

22.5.3 Class error_code

The class error_code describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [Note: Class error_code is an adjunct to error reporting by exception. —end note]

namespace std {

class error_code {

public:

// 22.5.3.2, constructors
error_code() noexcept;
error_code(int val, const error_category& cat) noexcept;
template<class ErrorCodeEnum>
error_code(ErrorCodeEnum e) noexcept;

// 22.5.3.3, modifiers
void assign(int val, const error_category& cat) noexcept;
template<class ErrorCodeEnum>
error_code& operator=(ErrorCodeEnum e) noexcept;
void clear() noexcept;

// 22.5.3.4, 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;

};

§ 22.5.3.1
private:
    int val_;    // exposition only
    const error_category* cat_; // exposition only
};

// 22.5.3.5, 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);

22.5.3.2 Class error_code constructors

error_code() noexcept;
    Effects: Constructs an object of type error_code.
    Postconditions: val_ == 0 and cat_ == &system_category().

error_code(int val, const error_category& cat) noexcept;
    Effects: Constructs an object of type error_code.
    Postconditions: val_ == val and cat_ == &cat.

template<class ErrorCodeEnum>
    error_code(ErrorCodeEnum e) noexcept;
    Effects: Constructs an object of type error_code.
    Postconditions: *this == make_error_code(e).
    Remarks: This constructor shall not participate in overload resolution unless
                is_error_code_enum_v<ErrorCodeEnum> is true.

22.5.3.3 Class error_code modifiers

void assign(int val, const error_category& cat) noexcept;
    Postconditions: val_ == val and cat_ == &cat.

template<class ErrorCodeEnum>
    error_code& operator=(ErrorCodeEnum e) noexcept;
    Effects: Constructs an object of type error_code.
    Postconditions: *this == make_error_code(e).
    Returns: *this.
    Remarks: This operator shall not participate in overload resolution unless
                is_error_code_enum_v<ErrorCodeEnum> is true.

void clear() noexcept;
    Postconditions: value() == 0 and category() == system_category().

22.5.3.4 Class error_code observers

int value() const noexcept;
    Returns: val_.

const error_category& category() const noexcept;
    Returns: *cat_.

erreur_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.

22.5.3.5  Class error_code non-member functions

   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: As if by: os << ec.category().name() << ':' << ec.value();

22.5.4  Class error_condition

22.5.4.1  Class error_condition 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 (22.5.3) are implementation specific. —end note]

   namespace std {
      class error_condition {
         public:
            // 22.5.4.2, constructors
            error_condition() noexcept;
            error_condition(int val, const error_category& cat) noexcept;
template<class ErrorConditionEnum>
            error_condition(ErrorConditionEnum e) noexcept;
         
            // 22.5.4.3, modifiers
            void assign(int val, const error_category& cat) noexcept;
template<class ErrorConditionEnum>
            error_condition& operator=(ErrorConditionEnum e) noexcept;
            void clear() noexcept;
         
            // 22.5.4.4, 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
      };
   }

22.5.4.2  Class error_condition constructors

   error_condition() noexcept;

   Effects: Constructs an object of type error_condition.

   Postconditions: val_ == 0 and cat_ == &generic_category().

   error_condition(int val, const error_category& cat) noexcept;

   Effects: Constructs an object of type error_condition.

   Postconditions: val_ == val and cat_ == &cat.

   template<class ErrorConditionEnum>
   error_condition(ErrorConditionEnum e) noexcept;

   Effects: Constructs an object of type error_condition.

   Postconditions: *this == make_error_condition(e).
Remarks: This constructor shall not participate in overload resolution unless
is_error_condition_enum_v<ErrorConditionEnum> is true.

22.5.4.3 Class error_condition modifiers

```cpp
void assign(int val, const error_category& cat) noexcept;
```

Postconditions: val_ == val and cat_ == &cat.

```cpp
template<class ErrorConditionEnum>
error_condition& operator=(ErrorConditionEnum e) noexcept;
```

Postconditions: *this == make_error_condition(e).

Returns: *this.

Remarks: This operator shall not participate in overload resolution unless
is_error_condition_enum_v<ErrorConditionEnum> is true.

```cpp
void clear() noexcept;
```

Postconditions: value() == 0 and category() == generic_category().

22.5.4.4 Class error_condition observers

```cpp
int value() const noexcept;
```

Returns: val_.

```cpp
const error_category& category() const noexcept;
```

Returns: *cat_.

```cpp
string message() const;
```

Returns: category().message(value()).

```cpp
explicit operator bool() const noexcept;
```

Returns: value() != 0.

22.5.4.5 Class error_condition non-member functions

```cpp
error_condition make_error_condition(errc e) noexcept;
```

Returns: error_condition(static_cast<int>(e), generic_category()).

22.5.5 Comparison functions

```cpp
bool operator<(const error_code& lhs, const error_code& rhs) noexcept;
```

Returns:

```cpp
(lhs.category() < rhs.category() ||
 (lhs.category() == rhs.category() && lhs.value() < rhs.value()))
```

```cpp
bool operator<(const error_condition& lhs, const error_condition& rhs) noexcept;
```

Returns:

```cpp
(lhs.category() < rhs.category() ||
 (lhs.category() == rhs.category() && lhs.value() < rhs.value()))
```

```cpp
bool operator==(const error_code& lhs, const error_code& rhs) noexcept;
```

Returns:

```cpp
(lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value()))
```

```cpp
bool operator==(const error_condition& lhs, const error_condition& rhs) noexcept;
```

Returns:

```cpp
(lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value()))
```

§ 22.5.5
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).

22.5.6 System error hash support

template<> struct hash<error_code>;
template<> struct hash<error_condition>;

The specializations are enabled (23.14.15).

22.5.7 Class system_error

22.5.7.1 Class system_error 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 (21.6.3.1) 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;
    };  
  }

22.5.7.2 Class system_error members

system_error(error_code ec, const string& what_arg);

Effects: Constructs an object of class system_error.
Postconditions: code() == ec and string(what()).find(what_arg) != string::npos.

system_error(error_code ec, const char* what_arg);

Effects: Constructs an object of class system_error.
Postconditions: code() == ec and string(what()).find(what_arg) != string::npos.

system_error(error_code ec);

Effects: Constructs an object of class system_error.
Postconditions: code() == ec.

§ 22.5.7.2
system_error(int ev, const error_category& ecat, const string& what_arg);

Effects: Constructs an object of class system_error.
Postconditions: code() == error_code(ev, ecat) and
                string(what()).find(what_arg) != string::npos.

system_error(int ev, const error_category& ecat, const char* what_arg);

Effects: Constructs an object of class system_error.
Postconditions: code() == error_code(ev, ecat) and
                string(what()).find(what_arg) != string::npos.

system_error(int ev, const error_category& ecat);

Effects: Constructs an object of class system_error.
Postconditions: code() == error_code(ev, ecat).

const error_code& code() const noexcept;

Returns: ec or error_code(ev, ecat), from the constructor, as appropriate.

const char* what() const noexcept override;

Returns: An NTBS incorporating the arguments supplied in the constructor.

[Note: The returned NTBS might be the contents of what_arg + ": " + code.message(). — end note]
23  General utilities library

23.1  General

This Clause describes utilities that are generally useful in C++ programs; some of these utilities are used by other elements of the C++ standard library. These utilities are summarized in Table 34.

Table 34 — General utilities library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.2  Utility components</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>23.3  Compile-time integer sequences</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>23.4  Pairs</td>
<td>&lt;utility&gt;</td>
</tr>
<tr>
<td>23.5  Tuples</td>
<td>&lt;tuple&gt;</td>
</tr>
<tr>
<td>23.6  Optional objects</td>
<td>&lt;optional&gt;</td>
</tr>
<tr>
<td>23.7  Variants</td>
<td>&lt;variant&gt;</td>
</tr>
<tr>
<td>23.8  Storage for any type</td>
<td>&lt;any&gt;</td>
</tr>
<tr>
<td>23.9  Fixed-size sequences of bits</td>
<td>&lt;bitset&gt;</td>
</tr>
<tr>
<td>23.10 Memory</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>23.11 Smart pointers</td>
<td>&lt;memory&gt;</td>
</tr>
<tr>
<td>23.12 Memory resources</td>
<td>&lt;memory_resource&gt;</td>
</tr>
<tr>
<td>23.13 Scoped allocators</td>
<td>&lt;scoped_allocator&gt;</td>
</tr>
<tr>
<td>23.14 Function objects</td>
<td>&lt;functional&gt;</td>
</tr>
<tr>
<td>23.15 Type traits</td>
<td>&lt;type_traits&gt;</td>
</tr>
<tr>
<td>23.16 Compile-time rational arithmetic</td>
<td>&lt;ratio&gt;</td>
</tr>
<tr>
<td>23.17 Time utilities</td>
<td>&lt;chrono&gt;</td>
</tr>
<tr>
<td>23.18 Type indexes</td>
<td>&lt;typeindex&gt;</td>
</tr>
<tr>
<td>23.19 Execution policies</td>
<td>&lt;execution&gt;</td>
</tr>
<tr>
<td>23.20 Primitive numeric conversions</td>
<td>&lt;charconv&gt;</td>
</tr>
</tbody>
</table>

23.2  Utility components

This subclause contains some basic function and class templates that are used throughout the rest of the library.

23.2.1  Header <utility> synopsis

```
#include <initializer_list>    // see 21.9.1

namespace std {
    // 23.2.2, swap
    template<class T>
    void swap(T& a, T& b) noexcept (see below);
    template<class T, size_t N>
    void swap(T (&a)[N], T (&b)[N]) noexcept (is_nothrow_swappable_v<T>);

    // 23.2.3, exchange
    template<class T, class U = T>
    constexpr T exchange(T& obj, U& new_val);

    // 23.2.4, 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;
```

§ 23.2.1
template<class T>
constexpr remove_reference_t<T>&& move(T&&) noexcept;

// 23.2.5, as_const
template<class T>
constexpr add_const_t<T>& as_const(T t) noexcept;
template<class T>
void as_const(const T& t) = delete;

// 23.2.6, declval
template<class T>
add_rvalue_reference_t<T> declval() noexcept;  // as unevaluated operand

// 23.3, 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 = make_integer_sequence<size_t, N>;

// 23.4, class template pair
template<class T1, class T2>
struct pair;

// 23.4.3, 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>
void swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y)));

// 23.4.4, tuple-like access to pair
template<class T>
struct tuple_size;
template<
size_t I, class T>
struct tuple_element;

template<class T1, class T2>
struct tuple_size<pair<T1, T2>>;
template<
size_t I, class T1, class T2>
struct tuple_element<pair<T1, T2>, I>;

template<
size_t I, class T1, class T2>
using get = tuple_element_t<I, pair<T1, T2>>&;

§ 23.2.1
The header `<utility>` defines several types and function templates that are described in this Clause. It also defines the template `pair` and various function templates that operate on `pair` objects.

23.2.2 `swap` [utility.swap]

```cpp
template<class T>
void swap(T& a, T& b) noexcept(see below);
```

1 Remarks: This function 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:

```cpp
is_nothrow_move_constructible_v<T> && is_nothrow_move_assignable_v<T>
```

2 Requires: Type `T` shall be `MoveConstructible` (Table 23) and `MoveAssignable` (Table 25).

3 Effects: Exchanges values stored in two locations.
template<class T, size_t N>
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 (20.5.3.2) b[i] for all i in the range [0, N).

Effects: As if by swap_ranges(a, a + N, b).

23.2.3 exchange

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;

23.2.4 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 (21.11.4).

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.

[Example:

```cpp
template<class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
    return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}
```

```cpp
struct A {
    A(int&, const double&);
};
```

void g() {
    shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&
    int i = 2;
    shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK
}

In the first call to factory, A1 is deduced as int, so 2 is forwarded to A's constructor as an rvalue. In the second call to factory, A1 is deduced as int&, so i is forwarded to A's constructor as an lvalue. In both cases, A2 is deduced as double, so 1.414 is forwarded to A's constructor as an rvalue. — end example]

template<class T> constexpr remove_reference_t<T>&& move(T&& t) noexcept;

Returns: static_cast<remove_reference_t<T>&&>(t).

[Example:

```cpp
template<class T, class A1>
shared_ptr<T> factory(A1&& a1) {
    return shared_ptr<T>(new T(std::forward<A1>(a1)));
}
```

```cpp
struct A {
    A();
    A(const A&); // copies from lvalues
    A(A&&); // moves from rvalues
};
```
void g() {
  A a;
  shared_ptr<A> sp1 = factory<A>(a);  // "a" binds to A(const A&)
  shared_ptr<A> sp1 = factory<A>(std::move(a));  // "a" binds to A(A&&)
}

In the first call to factory, A1 is deduced as A&, so a is forwarded as a non-const lvalue. This binds to
the constructor A(const A&), which copies the value from a. In the second call to factory, because of
the call std::move(a), A1 is deduced as A, so a is forwarded as an rvalue. This binds to the constructor
A(A&&), which moves the value from a. —end example

template<class T> constexpr conditional_t<
  !is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&>
movie_if_noexcept(T& x) noexcept;

Returns: std::move(x).

23.2.5 Function template as_const

template<class T> constexpr add_const_t<T>& as_const(T& t) noexcept;

Returns: t.

23.2.6 Function template declval

The library provides the function template declval to simplify the definition of expressions which occur as
unevaluated operands (8.2).

template<class T> add_rvalue_reference_t<T> declval() noexcept;

// as unevaluated operand

Remarks: If this function is odr-used (6.2), the program is ill-formed.

Remarks: The template parameter T of declval may be an incomplete type.

[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 (23.15.7.6). —end example]

23.3 Compile-time integer sequences

23.3.1 In general

The library provides a class template that can represent an integer sequence. When used as an argument to
a function template the 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 23.5.3.5. —end note]

23.3.2 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); }
  };
}

T shall be an integer type.

23.3.3 Alias template make_integer_sequence

template<class T, T N>
using make_integer_sequence = integer_sequence<T, see below>;

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]
23.4 Pairs

23.4.1 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 23.5.3.6 and 23.5.3.7).

23.4.2 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;
        EXPLICIT constexpr pair();
        EXPLICIT constexpr pair(const T1& x, const T2& y);
        template<class U1, class U2> EXPLICIT constexpr pair(U1&& x, U2&& y);
        template<class U1, class U2> EXPLICIT constexpr pair(const pair<U1, U2>& p);
        template<class U1, class U2> EXPLICIT constexpr pair(pair<U1, U2>&& p);
        template<class... Args1, class... Args2>
        pair(piecewise_construct_t, tuple<Args1...> first_args, tuple<Args2...> second_args);

        pair& operator=(const pair& p);
        template<class U1, class U2> pair& operator=(const pair<U1, U2>& p);
        pair& operator=(pair&& p) noexcept(see below);
        template<class U1, class U2> pair& operator=(pair<U1, U2>&& p);

        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. The destructor of pair shall be a trivial destructor if (is_trivially_destructible_v<T1> && is_trivially_destructible_v<T2>) is true.

EXPLICIT constexpr pair();

3 Effects: Value-initializes first and second.

4 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 constructor is explicit 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]

EXPLICIT constexpr pair(const T1& x, const T2& y);

5 Effects: Initializes first with x and second with y.

6 Remarks: This constructor shall not participate in overload resolution unless is_copy_constructible_v<const first_type&> is true and is_copy_constructible_v<const second_type&> is true. The constructor is explicit if and only if is_convertible_v<const first_type&, first_type> is false or is_convertible_v<const second_type&, second_type> is false.
template<class U1, class U2> EXPLICIT constexpr 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<first_type, U1&&> is true and is_constructible_v<second_type, U2&&> is true. The constructor is explicit if and only if is_convertible_v<U1&&, first_type> is false or is_convertible_v<U2&&, second_type> is false.

template<class U1, class U2> EXPLICIT constexpr 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<first_type, const U1&> is true and is_constructible_v<second_type, const U2&> is true. The constructor is explicit if and only if is_convertible_v<const U1&, first_type> is false or is_convertible_v<const U2&, second_type> is false.

template<class U1, class U2> EXPLICIT constexpr pair(pair<U1, U2>&& p);

Effects: Initializes first with std::forward<U1>(p.first) and second with std::forward<U2>(p.second).

Remarks: This constructor shall not participate in overload resolution unless is_constructible_<v<first_type, U1&&> is true and is_constructible_v<second_type, U2&&> is true. The constructor is explicit if and only if is_convertible_v<U1&&, first_type> is false or is_convertible_v<U2&&, second_type> is false.

template<class... Args1, class... Args2>
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.

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> pair& operator=(const pair<U1, U2>& p);

Effects: Assigns p.first to first and p.second to second.

Remarks: This operator shall not participate in overload resolution unless is_assignable_v<first_type&, const U1&> is true and is_assignable_v<second_type&, const U2&> is true.

Returns: *this.

pair& operator=(pair&& p) noexcept;

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<first_type> is true and is_move_assignable_v<second_type> is true.

Remarks: The expression inside noexcept is equivalent to:

is_nothrow_move_assignable_v<T1> && is_nothrow_move_assignable_v<T2>

Returns: *this.
template<class U1, class U2> pair& operator=(pair<U1, U2>&& p);

Effects: Assigns to first with std::forward<T>(p.first) and to second with std::forward<V>(p.second).

Remarks: This operator shall not participate in overload resolution unless is_assignable_v<first_type, U1&&> is true and is_assignable_v<second_type, U2&&> is true.

Returns: *this.

void swap(pair& p) noexcept(see below);

Requires: first shall be swappable with (20.5.3.2) 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:

is_nothrow_swappable_v<first_type> && is_nothrow_swappable_v<second_type>

23.4.3 Specialized algorithms [pairs.spec]

template<class T1, class T2>
constexpr bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);

Returns: x.first == y.first && x.second == y.second.

template<class T1, class T2>
constexpr 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: 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: !(y < x).

template<class T1, class T2> 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 is_swappable_v<T1> is true and is_swappable_v<T2> is true.

template<class T1, class T2>
constexpr pair<T1, T2> make_pair(T1&& x, T2&& y);

Returns: pair(x, y);

[Example: In place of:
  return pair<int, double>(5, 3.1415926); // explicit types

  a C++ program may contain:
    return make_pair(5, 3.1415926); // types are deduced]
23.4.4 Tuple-like access to pair

```
template<class T1, class T2>
struct tuple_size<pair<T1, T2>> : integral_constant<size_t, 2> { };
```

1 Requires: I < 2. The program is ill-formed if I is out of bounds.

2 Value: The type T1 if I == 0, otherwise the type T2.

```
template<size_t I, class T1, class T2>
constexpr tuple_element_t<I, pair<T1, T2>>& get(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.

```
template<class T1, class T2>
constexpr T1& get(pair<T1, T2>& p) noexcept;
```

4 Requires: T1 and T2 are distinct types. Otherwise, the program is ill-formed.

```
template<class T2, class T1>
constexpr T2& get(pair<T1, T2>& p) noexcept;
```

5 Returns: A reference to p.first.

```
template<class T1, class T2>
constexpr T2& get(pair<T1, T2>& p) noexcept;
```

6 Requires: T1 and T2 are distinct types. Otherwise, the program is ill-formed.

23.5 Tuples

23.5.1 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 23.4.

23.5.2 Header `<tuple>` synopsis

```cpp
namespace std {
  // 23.5.3, class template tuple
template<class... Types>
class tuple;

  // 23.5.3.4, tuple creation functions
inline constexpr unspecified ignore;
template<class... TTypes>
constexpr tuple<TTypes&&...> make_tuple(TTypes&&...);
template<class... TTypes>
constexpr tuple<TTypes&&...> forward_as_tuple(TTypes&&...) noexcept;
template<class... TTypes>
constexpr tuple<TTypes&...> tie(TTypes&...) noexcept;
template<class... Tuples>
constexpr tuple<TTypes...> tuple_cat(Tuples&&...);

  // 23.5.3.5, 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);

  // 23.5.3.6, tuple helper classes
template<class T> class tuple_size;
  // not defined
template<class T> class tuple_size<const T>;
template<class T> class tuple_size<volatile T>;
template<class T> class tuple_size<const volatile T>;
template<class... Types> class tuple_size<tuple<Types...>>;

  template<
    size_t I, class T,
    class... Types
  >
  constexpr tuple_element_t<I, tuple<Types...>> & get(tuple<Types...>&) noexcept;
  constexpr tuple_element_t<I, tuple<Types...>> && get(tuple<Types...>&&) noexcept;
  constexpr const tuple_element_t<I, tuple<Types...>> & get(const tuple<Types...>&) noexcept;
  constexpr const tuple_element_t<I, tuple<Types...>> && get(const tuple<Types...>&&) noexcept;
  using tuple_element_t = typename tuple_element<I, T>::type;

  // 23.5.3.7, element access
  template<
    size_t I, class... Types
  >
  constexpr tuple_element_t<I, tuple<Types...>>& get(tuple<Types...>&) noexcept;
  constexpr tuple_element_t<I, tuple<Types...>>&& get(tuple<Types...>&&) noexcept;
  constexpr const tuple_element_t<I, tuple<Types...>>& get(const tuple<Types...>&) noexcept;
  constexpr const tuple_element_t<I, tuple<Types...>>&& get(const tuple<Types...>&&) 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;
```
```cpp
// 23.5.3.8, relational operators
template<class... TTypes, class... UTypes>
constexpr bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);  // only if sizeof...(TTypes) == 1

// 23.5.3.9, allocator-related traits
template<class... Types, class Alloc>
struct uses_allocator<tuple<Types...>, Alloc>;

// 23.5.3.10, specialized algorithms
template<class... Types>
void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);

// 23.5.3.6, tuple helper classes
template<class T>
inline constexpr size_t tuple_size_v = tuple_size<T>::value;
```

## 23.5.3 Class template `tuple`

```cpp
template<class... Types>
class tuple {
  public:
  // 23.5.3.1, tuple construction
  EXPLICIT constexpr tuple();
  EXPLICIT constexpr tuple(const Types&...);  // only if sizeof...(Types) == 1
  template<class... UTypes>
  EXPLICIT constexpr tuple(UTypes&&...);  // only if sizeof...(Types) == 1

  tuple(const tuple&) = default;
  tuple(tuple&&) = default;

  template<class... UTypes>
  EXPLICIT constexpr tuple(const tuple<UTypes...>&);  
  template<class... UTypes>
  EXPLICIT constexpr tuple(tupule<UTypes...>&&);

  template<class U1, class U2>
  EXPLICIT constexpr tuple(const pair<U1, U2>&);  // only if sizeof...(Types) == 2
  template<class U1, class U2>
  EXPLICIT constexpr tuple(pair<U1, U2>&&);  // only if sizeof...(Types) == 2

  // allocator-extended constructors
  template<class Alloc>
  tuple(allocator_arg_t, const Alloc& a);
  template<class Alloc>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, const Types&...);
  template<class Alloc, class... UTypes>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, UTypes&...);
```
template<class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const tuple&);
template<class Alloc>
  tuple(allocator_arg_t, const Alloc& a, tuple&&);

template<class Alloc, class... UTypes>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template<class Alloc, class... UTypes>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);

template<class Alloc, class U1, class U2>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, const pair<U1, U2>&);
template<class Alloc, class U1, class U2>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, pair<U1, U2>&&);

// 23.5.3.2, tuple assignment
tuple& operator=(const tuple&);
tuple& operator=(tuple&&) noexcept;

#define DEFAULT_MOVE_U1

template<class... UTypes>
  tuple& operator=(const tuple<UTypes...>&);
template<class... UTypes>
  tuple& operator=(tuple<UTypes...>&&);

#define DEFAULT_MOVE_U2

template<class U1, class U2>
  tuple& operator=(const pair<U1, U2>&);

#define DEFAULT_MOVE_U3

template<class U1, class U2>
  tuple& operator=(pair<U1, U2>&&);

#define DEFAULT_MOVE_U4

// 23.5.3.3, tuple swap

void swap(tuple&) noexcept;

};

template<class... UTypes>
  tuple(UTypes...) -> tuple<UTypes...>;

template<class T1, class T2>
  tuple(pair<T1, T2>) -> tuple<T1, T2>;

template<class Alloc, class... UTypes>
  tuple(allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;

template<class Alloc, class T1, class T2>
  tuple(allocator_arg_t, Alloc, pair<T1, T2>) -> tuple<T1, T2>;

template<class Alloc, class... UTypes>
  tuple(allocator_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>;

23.5.3.1 Construction [tuple.cnstr]

1 For each tuple constructor, an exception is thrown only if the construction of one of the types in Types throws an exception.

2 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.

3 The destructor of tuple shall be a trivial destructor if (is_trivially_destructible_v<TTypes> && ...) is true.

4 In the constructor descriptions that follow, let \( i \) be in the range \([0, \text{sizeof}(\ldots)\text{Types})\) in order, \( T_i \) be the \( i \)th type in Types, and \( U_i \) be the \( i \)th type in a template parameter pack named UTypes, where indexing is zero-based.

EXPLICIT constexpr tuple();

Effects: Value-initializes each element.

Remarks: This constructor shall not participate in overload resolution unless is_default_constructible_v<T\( T_i \)> is true for all \( i \). [Note: This behavior can be implemented by a constructor template with default template arguments. — end note] The constructor is explicit 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 \(\text{const T}_i\)\& can be initialized with \(\{\}\). — end note]

**EXPLICIT constexpr** `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)` \(\geq 1\) and `is_copy_constructible_v<T_i>` is `true` for all \(i\). The constructor is explicit if and only if `is_convertible_v<\text{const T}_i, T_i>` is `false` for at least one \(i\).

```cpp
template<class... UTiles> EXPLICIT constexpr tuple(UTiles&&... u);
```

*Effects:* Initializes the elements in the tuple with the corresponding value in `std::forward<UTiles>(u)`.

*Remarks:* This constructor shall not participate in overload resolution unless `sizeof...(Types)` \(= sizeof...(UTiles)` and `is_convertible_v<\text{const T}_i, U_i,\&>` is `true` for all \(i\). The constructor is explicit if and only if `is_convertible_v<\text{const T}_i, T_i>` is `false` for at least one \(i\).

```cpp
tuple(const tuple& u) = default;
```

*Requires:* `is_copy_constructible_v<T_i>` is `true` for all \(i\).

*Effects:* Initializes each element of `*this` with the corresponding element of `u`.

```cpp
tuple(tuple& u) = default;
```

*Requires:* `is_move_constructible_v<T_i>` is `true` for all \(i\).

*Effects:* For all \(i\), initializes the \(i^{th}\) element of `*this` with `std::forward<T_i>(get<i>(u))`.

```cpp
template<class... UTiles> EXPLICIT constexpr tuple(const tuple<UTiles...>& u);
```

*Effects:* Initializes each element of `*this` with the corresponding element of `u`.

*Remarks:* This constructor shall not participate in overload resolution unless

(16.1) `sizeof...(Types) = sizeof...(UTiles)` and

(16.2) `is_constructible_v<T_i, const U_i,\&>` is `true` for all \(i\), and

(16.3) either `sizeof...(Types)` \(\neq 1\), or (when `Types...` expands to `T` and `UTiles...` expands to `U`) `is_convertible_v<\text{const tuple<U>&, T_i, is_constructible_v<\text{const tuple<U>&, T_i, is_same_v<T, U>>}, and is_same_v<T, U>` are all `false`.

The constructor is explicit if and only if `is_convertible_v<\text{const U_i, T_i}>` is `false` for at least one \(i\).

```cpp
template<class... UTiles> EXPLICIT constexpr tuple(tuple<UTiles...>&& u);
```

*Effects:* For all \(i\), initializes the \(i^{th}\) element of `*this` with `std::forward<U_i>(get<i>(u))`.

*Remarks:* This constructor shall not participate in overload resolution unless

(18.1) `sizeof...(Types) = sizeof...(UTiles)` and

(18.2) `is_constructible_v<T_i, U_i,\&>` is `true` for all \(i\), and

(18.3) either `sizeof...(Types)` \(\neq 1\), or (when `Types...` expands to `T` and `UTiles...` expands to `U`) `is_convertible_v<\text{tuple<U>, T_i}, is_constructible_v<\text{tuple<U>, T_i}, is_same_v<T, U>` are all `false`.

The constructor is explicit if and only if `is_convertible_v<U_i,\&>, T_i>` is `false` for at least one \(i\).

```cpp
template<class U1, class U2> EXPLICIT constexpr tuple(const pair<U1, U2>& u);
```

*Effects:* Initializes the first element with `u.first` and the second element with `u.second`.

*Remarks:* This constructor shall not participate in overload resolution unless `sizeof...(Types)` \(= 2\), `is_constructible_v<T_0, const U1,\&>` is `true` and `is_constructible_v<T_1, const U2,\&>` is `true`.

The constructor is explicit if and only if `is_convertible_v<\text{const U1, T_0}>` is `false` or `is_convertible_v<\text{const U2, T_1}>` is `false`.

§ 23.5.3.1
template<class U1, class U2> EXPLICIT constexpr tuple(pair<U1, U2>&& u);

Effects: Initializes the first element with std::forward<U1>(u.first) and the second element with std::forward<U2>(u.second).

Remarks: This constructor shall not participate in overload resolution unless sizeof...(Types) == 2, is_convertible_v<T0, U1&&> is true and is_convertible_v<T1, U2&&> is true. The constructor is explicit if and only if is_convertible_v<U1&&, T0> is false or is_convertible_v<U2&&, T1> is false.

template<class Alloc>
  tuple(allocator_arg_t, const Alloc& a);
template<class Alloc>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, const Types&...);
template<class Alloc, class... UTypes>
  EXPLICIT tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template<class Alloc>
  tuple(allocator_arg_t, const Alloc& a, const tuple&);
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 is_copyAssignable_v<Ti> is true for all i.
Returns: *this.

tuple& operator=(tuple&& u) noexcept(see below);

Effects: For all i, assigns std::forward<Ti>(get<i>(u)) to get<i>(*this).
Remarks: This operator shall not participate in overload resolution unless is_moveAssignable_v<Ti> is true for all i.
Remarks: The expression inside noexcept is equivalent to the logical AND of the following expressions:
  is_nothrow_moveAssignable_v<Ti>
where Ti is the i th type in Types.
Returns: *this.

template<class... UTypes> tuple& operator=(const tuple<UTypes...>& u);

Effects: Assigns each element of u to the corresponding element of *this.
Remarks: This operator shall not participate in overload resolution unless sizeof...(Types) == sizeof...(UTypes) and isAssignable_v<Ti, const Ui&> is true for all i.
Returns: *this.
template<class... UTypes> tuple& operator=(tuple<UTypes...>&& u);

Effects: For all \(i\), assigns `std::forward<U_i>(get<i>(u))` to `get<i>(*this)`.
Remarks: This operator shall not participate in overload resolution unless `is_assignable_v<T_i&, U_i&&>` is true for all \(i\) and `sizeof...(Types) == sizeof...(UTypes)`.
Returns: *this.

template<class U1, class U2> tuple& operator=(const pair<U1, U2>& u);

Effects: Assigns `u.first` to the first element of *this and `u.second` to the second element of *this.
Remarks: This operator shall not participate in overload resolution unless `sizeof...(Types) == 2` and `is_assignable_v<T_0&, const U1&>` is true for the first type \(T_0\) in Types and `is_assignable_v<T_1&, const U2&>` is true for the second type \(T_1\) in Types.
Returns: *this.

23.5.3.3 swap [tuple.swap]

```cpp
void swap(tuple& rhs) noexcept(see below);
```

Requires: Each element in *this shall be swappable with (20.5.3.2) the corresponding element in rhs.
Effects: Calls `swap` for each element in *this and its corresponding element in rhs.
Remarks: The expression inside `noexcept` is equivalent to the logical and of the following expressions:
`is_nothrow_swappable_v<T_i>`
where \(T_i\) is the \(i\)th type in Types.
Throws: Nothing unless one of the element-wise `swap` calls throws an exception.

23.5.3.4 Tuple creation functions [tuple.creation]

In the function descriptions that follow, the members of a parameter pack `XTypes` are denoted by `X_i` for \(i\) in \([0, sizeof...(XTypes))\) in order, where indexing is zero-based.

```cpp
template<class... TTypes>
constexpr tuple<TTypes...> make_tuple(TTypes&&... t);
```

The pack `VTypes` is defined as follows. Let \(U_i\) be `decay_t<T_i>` for each \(T_i\) in `TTypes`. If \(U_i\) is a specialization of `reference_wrapper`, then \(V_i\) in `VTypes` is \(U_i::type&\), otherwise \(V_i\) is \(U_i\).
Returns: `tuple<VTypes...>(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]

```cpp
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:

```cpp
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
—end example]

template<class... Tuples>
constexpr tuple<CTypes...> tuple_cat(Tuples&&... tpls);

In the following paragraphs, let Ti be the ith type in Tuples, Ui be remove_reference<Ti>, and tp_i be the ith parameter in the function parameter pack tpls, where all indexing is zero-based.

Requires: For all i, Ui shall be the type cv_i tuple<Args...>, where cv_i is the (possibly empty) ith cv-qualifier-seq and Args_i is the parameter pack representing the element types in Ui. Let Ai_k be the kth type in Args_i. For all Ai_k the following requirements shall be satisfied:

(10.1) — If Ti is deduced as an lvalue reference type, then is_constructible_v<A_i_k, cv_i A_i_k&> == true, otherwise

(10.2) — 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_n-1, ..., Args_n, where n is equal to sizeof...(Tuples). Let e_i... be the ith 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 kth type element e_i_k in e_i... with get<k_i>(std::forward<Ti>(tp_i)) for each valid k_i and each group e_i in order.

[Note: An implementation may support additional types in the parameter pack Tuples that support the tuple-like protocol, such as pair and array. —end note]

23.5.3.5 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:

```cpp
template<class F, class Tuple, size_t... I>
constexpr decltype(auto) apply_impl(F&& f, Tuple&& t, index_sequence<I...>) {
// exposition only
  return INVOKE(std::forward<F>(f), std::get<I>(std::forward<Tuple>(t))...);  // see 23.14.3
}
```

Equivalent to:

```cpp
return apply_impl(std::forward<F>(f), std::forward<Tuple>(t),
  make_index_sequence<tuple_size_v<remove_reference<Tuple>>>{});
```

23.5.3.5 Calling a function with a tuple of arguments [tuple.apply]
return make_from_tuple_impl<T>(
forward<Tuple>(t),
make_index_sequence<tuple_size_v<remove_reference_t<Tuple>>>{});

[Note: The type of T must be supplied as an explicit template parameter, as it cannot be deduced from the argument list. — end note]

23.5.3.6 Tuple helper classes

template<class T> struct tuple_size;

Remarks: All specializations of tuple_size shall meet the UnaryTypeTrait requirements (23.15.1) with a base characteristic of integral_constant<size_t, N> for some N.

template<class... Types>
class tuple_size<tuple<Types...>> : public integral_constant<size_t, sizeof...(Types)>{ };

template<size_t I, class... Types>
class tuple_element<I, tuple<Types...>> { 
public:
  using type = TI;
};

Requires: I < sizeof...(Types). The program is ill-formed if I is out of bounds.

Type: TI is the type of the Ith element of Types, where indexing is zero-based.

template<class T> class tuple_size<const T>;
template<class T> class tuple_size<volatile T>;
template<class T> class tuple_size<const volatile T>;

Let TS denote tuple_size<T> of the cv-unqualified type T. If the expression TS::value is well-formed when treated as an unevaluated operand, then each of the three templates shall meet the UnaryTypeTrait requirements (23.15.1) with a base characteristic of

integral_constant<size_t, TS::value>

Otherwise, they shall have no member value.

Access checking is performed as if in a context unrelated to TS and T. Only the validity of the immediate context of the expression is considered. [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 <tuple> header, the three templates are available when either of the headers <array> or <utility> are included.

template<size_t I, class T> class tuple_element<I, const T>;
template<size_t I, class T> class tuple_element<I, volatile T>;
template<size_t I, class T> class tuple_element<I, const volatile T>;

Let TE denote tuple_element_t<I, T> of the cv-unqualified type T. Then each of the three templates shall meet the TransformationTrait requirements (23.15.1) with a member typedef type that names the following type:

— for the first specialization, add_const_t<TE>,
— for the second specialization, add_volatile_t<TE>, and
— 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 either of the headers <array> or <utility> are included.

23.5.3.7 Element access

template<size_t I, class... Types>
constexpr tuple_element_t<I, tuple<Types...>>&
get(tuple<Types...>& t) noexcept;

§ 23.5.3.7 503
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

Requires: I < sizeof...(Types). The program is ill-formed if I is out of bounds.
Returns: A reference to the Ith element of t, where indexing is zero-based.

[Note A: If a T in 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 T in Types is some reference type X&, the return type is X&, not
    const X&. However, if the element type is a non-reference type T, the return type is 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 Types.... Otherwise, the program is ill-formed.
Returns: A reference to the element of t corresponding to the type T in Types....

[Example:
    const tuple<int, const int, double, double> t(1, 2, 3.4, 5.6);
    const int& i1 = get<int>(t); // OK. 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 example]

[Note: The reason get is a non-member function is that if this functionality had been provided as a member
    function, code where the type depended on a template parameter would have required using the template
    keyword. —end note]

23.5.3.8 Relational operators [tuple.rel]

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 < sizeof...(TTypes), get<i>(t) == get<i>(u) is a valid
    expression returning a type that is convertible to bool. sizeof...(TTypes) == sizeof...(UTypes).
Returns: true if get<i>(t) == get<i>(u) for all i, otherwise false. For any two zero-length tuples
    e and f, e == f returns true.
Effects: The elementary comparisons are performed in order from the zeroth index upwards. No
    comparisons or element accesses are performed after the first equality comparison that evaluates to
    false.

template<class... TTypes, class... UTypes>
constexpr bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);

Requires: For all i, where 0 <= i and i < sizeof...(TTypes), both get<i>(t) < get<i>(u) and
    get<i>(u) < get<i>(t) are valid expressions returning types that are convertible to bool.
    sizeof...(TTypes) == sizeof...(UTypes).
Returns: The result of a lexicographical comparison between t and u. The result is defined as:
    (bool)(get<0>(t) < get<0>(u)) || (!bool)(get<0>(u) < get<0>(t)) && t.tail < u.tail), where

§ 23.5.3.8 504
\textit{r}_{\text{tail}}$ for some tuple \textit{r} is a tuple containing all but the first element of \textit{r}. For any two zero-length tuples \textit{e} and \textit{f}, \textit{e} < \textit{f} returns \textit{false}.  

\begin{verbatim}
template<class... TTypes, class... UTypes>
  constexpr bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
Returns: !\textit{(t == u)}.  

template<class... TTypes, class... UTypes>
  constexpr bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
Returns: \textit{u < t}.  

template<class... TTypes, class... UTypes>
  constexpr bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
Returns: !\textit{(u < t)}.  

Note: The above definitions for comparison functions do not require \textit{t}_{\text{tail}} (or \textit{u}_{\text{tail}}) to be constructed. It may not even be possible, as \textit{t} and \textit{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]  
\end{verbatim}

23.5.3.9 Tuple traits [tuple.traits]  

\begin{verbatim}
template<class... Types, class Alloc>
  struct uses_allocator<tuple<Types...>, Alloc> : true_type { };  
Returns: Alloc shall be an Allocator (20.5.3.5).  

Note: Specialization of this trait informs other library components that \textit{tuple} can be constructed with an allocator, even though it does not have a nested allocator\_type. —end note]  
\end{verbatim}

23.5.3.10 Tuple specialized algorithms [tuple.special]  

\begin{verbatim}
template<class... Types>
  void swap(tuple<Types...>& x, tuple<Types...>& y) noexcept(see below);
Remarks: This function shall not participate in overload resolution unless \textit{is\_swappable\_v} is \textit{true} for all \textit{i}, where \textit{0} \leq \textit{i} < \textit{sizeof...(Types)}. The expression inside noexcept is equivalent to:  
  noexcept(x.swap(y))  
Effects: As if by \textit{y}.swap(x).  
\end{verbatim}

23.6 Optional objects [optional]  

23.6.1 In general [optional.general]  

This subclause describes class template \textit{optional} that represents optional objects. An \textit{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.  

23.6.2 Header <optional> synopsis [optional.syn]  

namespace std {
  // 23.6.3, class template optional
  template<class T>
    class optional;

  // 23.6.4, no-value state indicator
  struct nullopt_t(see below);
  inline constexpr nullopt_t nullopt(unspecified);

§ 23.6.2
// 23.6.5, class bad_optional_access
class bad_optional_access;

// 23.6.6, 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>&);

// 23.6.7, 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;

// 23.6.8, 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>&);

// 23.6.9, specialized algorithms
template<class T>
void swap(optional<T>&, optional<T>&) noexcept(see below);

template<class T>
constexpr optional<see below> make_optional(T&&);
template<class T, class... Args>
constexpr optional<T> make_optional(Arg&&... args);
template<class T, class U, class... Args>
constexpr optional<T> make_optional(initializer_list<U> il, Arg&&... args);

// 23.6.10, hash support
template<class T> struct hash;
template<class T> struct hash(optional<T>);

1 A program that necessitates the instantiation of template optional for a reference type, or for possibly cv-qualified types in_place_t or nullopt_t is ill-formed.
23.6.3 Class template optional

```cpp
template<class T>
class optional {
public:
  using value_type = T;

  // 23.6.3.1, constructors
  constexpr optional() noexcept;
  constexpr optional(nullopt_t) noexcept;
  constexpr optional(const optional&);
  constexpr optional(optional&&) noexcept;
  template<class... Args>
    constexpr explicit optional(in_place_t, Args&&...);
  template<class U, class... Args>
    constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
  template<class U = T>
    EXPLICIT constexpr optional(U&&);
  template<class U>
    EXPLICIT optional(const optional<U>&);
  template<class U>
    EXPLICIT optional(optional<U>&&);

  // 23.6.3.2, destructor
  ~optional();

  // 23.6.3.3, assignment
  optional& operator=(nullopt_t) noexcept;
  optional& operator=(const optional&);
  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&&...);

  // 23.6.3.4, swap
  void swap(optional&) noexcept;

  // 23.6.3.5, 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&;
  template<class U> constexpr T value_or(U&&) &&;

  // 23.6.3.6, modifiers
  void reset() noexcept;

private:
  T *val; // exposition only
};

template<class T> optional(T) -> optional<T>;
```

§ 23.6.3
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 and shall satisfy the requirements of `Destructible` (Table 27).

### 23.6.3.1 Constructors

**constexpr optional() noexcept;**

**constexpr optional(nullopt_t) noexcept;**

1. **Postconditions:** `*this` does not contain a value.
2. **Remarks:** No contained value is initialized. For every object type `T` these constructors shall be constexpr constructors (10.1.5).

**constexpr optional(const optional& rhs);**

3. **Effects:** If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs`.
4. **Postconditions:** `bool(rhs) == bool(*this)`.
5. **Throws:** Any exception thrown by the selected constructor of `T`.
6. **Remarks:** This constructor shall be defined as deleted unless `is_copy_constructible_v<T>` is `true`. If `is_trivially_copy_constructible_v<T>` is `true`, this constructor shall be a `constexpr` constructor.

**constexpr optional(optional&& rhs) noexcept(see below);**

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. **Postconditions:** `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 shall be a `constexpr` constructor.

**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. **Postconditions:** `*this` contains a value.
13. **Throws:** Any exception thrown by the selected constructor of `T`.
14. **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`.

**template<class U, class... Args> constexpr explicit optional(in_place_t, initializer_list<U> il, Args&&... args);**

15. **Effects:** Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.
16. **Postconditions:** `*this` contains a value.
17. **Throws:** Any exception thrown by the selected constructor of `T`.
Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is true. If T's constructor selected for the initialization is a constexpr constructor, this constructor shall be a constexpr constructor.

[Note: The following constructors are conditionally specified as explicit. This is typically implemented by declaring two such constructors, of which at most one participates in overload resolution. —end note]

template<class U = T> EXPLICIT 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)`.

Postconditions: `*this` contains a value.

Throws: Any exception thrown by the selected constructor of T.

Remarks: If T's selected constructor is a constexpr constructor, this constructor 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<U>, optional>` is false. The constructor is explicit if and only if `is_convertible_v<U&&, T>` is false.

template<class U> EXPLICIT optional(const 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 `*rhs`.

Postconditions: `bool(rhs) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of T.

Remarks: This constructor shall not participate in overload resolution unless

(27.1) `is_constructible_v<T, const U&>` is true,
(27.2) `is_constructible_v<T, optional<U>>&>` is false,
(27.3) `is_constructible_v<T, optional<U>&>` is false,
(27.4) `is_constructible_v<T, const optional<U>&>` is false,
(27.5) `is_constructible_v<T, const optional<U>&&>` is false,
(27.6) `is_convertible_v<optional<U>&, T>` is false,
(27.7) `is_convertible_v<optional<U>&&`, T` is false,
(27.8) `is_convertible_v<optional<U>&, T>` is false, and
(27.9) `is_convertible_v<optional<U>&&`, T` is false.

The constructor is explicit if and only if `is_convertible_v<T, const U&, T>` is false.

template<class U> EXPLICIT 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.

Postconditions: `bool(rhs) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of T.

Remarks: This constructor shall not participate in overload resolution unless

(31.1) `is_constructible_v<T, U&&>` is true,
(31.2) `is_constructible_v<T, optional<U>&>` is false,
(31.3) `is_constructible_v<T, optional<U>&&>` is false,
(31.4) `is_constructible_v<T, const optional<U>&>` is false,
(31.5) `is_constructible_v<T, const optional<U>&&>` is false,
(31.6) `is_convertible_v<optional<U>&, T>` is false,
(31.7) `is_convertible_v<optional<U>&&`, T` is false,
(31.8) `is_convertible_v<optional<U>&, T>` is false, and

§ 23.6.3.1
is_convertible_v<const optional<U>&&, T> is false.

The constructor is explicit if and only if is_convertible_v<U&&, T> is false.

### 23.6.3.2 Destructor

```cpp
~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> == true then this destructor shall be a trivial destructor.

### 23.6.3.3 Assignment

```cpp
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.

**Postconditions:** *this does not contain a value.

```cpp
optional<T>& operator=(const optional& rhs);
```

**Effects:** See Table 35.

**Table 35 — optional::operator=(const optional&) effects**

<table>
<thead>
<tr>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>assigns *rhs to the contained value</td>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>destroys the contained value by calling <code>val-&gt;T::~T()</code></td>
</tr>
</tbody>
</table>

**Returns:** *this.

**Postconditions:** 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.

```cpp
optional<T>& operator=(optional&& rhs) noexcept;
```

**Effects:** See Table 36. The result of the expression bool(rhs) remains unchanged.

**Table 36 — optional::operator=(optional&&) effects**

<table>
<thead>
<tr>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>assigns std::move(*rhs) to the contained value</td>
</tr>
<tr>
<td>rhs does not contain a value</td>
<td>destroys the contained value by calling <code>val-&gt;T::~T()</code></td>
</tr>
</tbody>
</table>

**Returns:** *this.
Postconditions: \( \text{bool}(\text{rhs}) == \text{bool}(\ast\text{this}) \).

Remarks: The expression inside \texttt{noexcept} is equivalent to:

\[ \text{is\_nothrow\_move\_assignable\_v<T> } \&\& \text{ is\_nothrow\_move\_constructible\_v<T>} \]

If any exception is thrown, the result of the expression \( \text{bool}(\ast\text{this}) \) remains unchanged. If an exception is thrown during the call to \( T \)'s move constructor, the state of \( \ast\text{rhs}.\text{val} \) 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 \( \ast\text{val} \) and \( \ast\text{rhs}.\text{val} \) is determined by the exception safety guarantee of \( T \)'s move assignment. This operator shall not participate in overload resolution unless \( \text{is\_move\_constructible\_v<T>} \) is true and \( \text{is\_move\_assignable\_v<T>} \) is true.

```cpp
template<class U = T> optional<T>& operator=(U&& v);
```

Effects: If \( \ast\text{this} \) contains a value, assigns \( \text{std::forward}<U>(v) \) to the contained value; otherwise initializes the contained value as if direct-non-list-initializing object of type \( T \) with \( \text{std::forward}<U>(v) \).

Returns: \( \ast\text{this} \).

Postconditions: \( \ast\text{this} \) contains a value.

Remarks: If any exception is thrown, the result of the expression \( \text{bool}(\ast\text{this}) \) 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 \( \ast\text{val} \) and \( v \) is determined by the exception safety guarantee of \( T \)'s assignment. This function shall not participate in overload resolution unless \( \text{is\_same}\_v<\text{remove\_cvref\_t<U>, optional}> \) is false, \( \text{conjunction\_v<is\_scalar<T>, is\_same<T, \text{decay\_t<U>>}> is false, is\_constructible\_v<T,U> is true, and is\_assignable\_v<T&, U> is true.} \)

```cpp
template<class U> optional<T>& operator=(const optional<U>& rhs);
```

Effects: See Table 37.

<table>
<thead>
<tr>
<th></th>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rhs}) contains a value</td>
<td>assigns ( \ast\text{rhs} ) to the contained value</td>
<td>initializes the contained value as if direct-non-list-initializing an object of type ( T ) with ( \ast\text{rhs} )</td>
</tr>
<tr>
<td>( \text{rhs}) does not contain a value</td>
<td>destroys the contained value by calling ( \text{val}\rightarrow\text{T}::\text{T}() )</td>
<td>no effect</td>
</tr>
</tbody>
</table>

Returns: \( \ast\text{this} \).

Postconditions: \( \text{bool}(\text{rhs}) == \text{bool}(\ast\text{this}) \).

Remarks: If any exception is thrown, the result of the expression \( \text{bool}(\ast\text{this}) \) remains unchanged. If an exception is thrown during the call to \( T \)'s constructor, the state of \( \ast\text{rhs}.\text{val} \) 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 \( \ast\text{val} \) and \( \ast\text{rhs}.\text{val} \) is determined by the exception safety guarantee of \( T \)'s assignment. This function shall not participate in overload resolution unless

\[
\begin{align*}
(20.1) & \quad \text{is\_constructible\_v<T, const U&> is true}, \\
(20.2) & \quad \text{is\_assignable\_v<T&, const U&> is true}, \\
(20.3) & \quad \text{is\_constructible\_v<T, optional<U>&> is false}, \\
(20.4) & \quad \text{is\_constructible\_v<T, optional<U>&&> is false}, \\
(20.5) & \quad \text{is\_constructible\_v<T, const optional<U>&> is false}, \\
(20.6) & \quad \text{is\_constructible\_v<T, const optional<U>&&> is false}, \\
(20.7) & \quad \text{is\_convertible\_v<optional<U>&, T> is false}, \\
(20.8) & \quad \text{is\_convertible\_v<optional<U>&&, T> is false}, \\
(20.9) & \quad \text{is\_convertible\_v<const optional<U>&, T> is false}, \\
\end{align*}
\]


\[
is_{\text{convertible}}_{\text{v}<\text{const optional}<U>&&, T> \text{ is false,}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{optional}<U>&> \text{ is false,}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{optional}<U>&&} \text{ is false,}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{const optional}<U>&>} \text{ is false, and}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{const optional}<U>&&} \text{ is false.}
\]

\[\text{template}<\text{class } U> \text{ optional}<T&> \text{ operator=}(<\text{optional}<U>&& \text{ rhs});\]

\textit{Effects:} See Table 38. The result of the expression \(\text{bool}(\text{rhs})\) remains unchanged.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{rhs contains a value} & \textbf{*this does not contain a value} \\
\hline
\textit{*this} contains a value & assigns \texttt{std::move(*rhs)} to the contained value \hline
\textit{rhs} does not contain a value & destroys the contained value by calling \texttt{val->T::~T()} \hline
\end{tabular}
\end{table}

\textit{Returns: *this.}

\textit{Postconditions: bool(rhs) == bool(*this).}

\textit{Remarks:} If any exception is thrown, the result of the expression \(\text{bool}(*\text{this})\) remains unchanged. If an exception is thrown during the call to \(T\)'s constructor, the state of \(*\text{rhs}.\text{val}\) 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 \(*\text{val}\) and \(*\text{rhs}.\text{val}\) is determined by the exception safety guarantee of \(T\)'s assignment. This function shall not participate in overload resolution unless

\[
is_{\text{constructible}}_{\text{v}<T, U> \text{ is true,}}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, U> \text{ is true,}}
\]

\[
is_{\text{constructible}}_{\text{v}<T, \text{optional}<U>&> \text{ is false,}}
\]

\[
is_{\text{constructible}}_{\text{v}<T, \text{optional}<U>&&} \text{ is false,}
\]

\[
is_{\text{constructible}}_{\text{v}<T, \text{const optional}<U>&>} \text{ is false,}
\]

\[
is_{\text{constructible}}_{\text{v}<T, \text{const optional}<U>&&} \text{ is false,}
\]

\[
is_{\text{convertible}}_{\text{v}<\text{optional}<U>&&, T> \text{ is false,}}
\]

\[
is_{\text{convertible}}_{\text{v}<\text{optional}<U>&&}, T> \text{ is false,}
\]

\[
is_{\text{convertible}}_{\text{v}<\text{const optional}<U>&, T> \text{ is false,}}
\]

\[
is_{\text{convertible}}_{\text{v}<\text{const optional}<U>&&}, T> \text{ is false,}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{optional}<U>&> \text{ is false,}}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{optional}<U>&&} \text{ is false,}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{const optional}<U>&> \text{ is false, and}
\]

\[
is_{\text{assignable}}_{\text{v}<T&, \text{const optional}<U>&&} \text{ is false.}
\]

\[\text{template}<\text{class... Args}> T& \text{ emplace(Args&&... args);}\]

\textit{Requires: is\_constructible\_v<T, Args&&...> is true.}

\textit{Effects:} Calls \texttt{*this = nullopt}. Then initializes the contained value as if direct-non-list-initializing an object of type \(T\) with the arguments \texttt{std::forward<Args>(args)}... .

\textit{Postconditions: *this contains a value.}

\textit{Returns:} A reference to the new contained value.

\textit{Throws:} Any exception thrown by the selected constructor of \(T\).
Remarks: If an exception is thrown during the call to T’s constructor, *this does not contain a value, and the previous *val (if any) has been destroyed.

template<class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);

Effects: Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Args>(args)....

Postconditions: *this contains a value.

Returns: A reference to the new contained value.

Remarks: If an exception is thrown during the call to T’s constructor, *this does not contain a value, and the previous *val (if any) has been destroyed. This function shall not participate in overload resolution unless is_constructible_v<T, initializer_list<U>&, Args&&...> is true.

23.6.3.4 Swap [optional.swap]

void swap(optional& rhs) noexcept(see below);

Requires: Lvalues of type T shall be swappable and is_move_constructible_v<T> is true.

Effects: See Table 39.

Table 39 — optional::swap(optional&) effects

<table>
<thead>
<tr>
<th></th>
<th>*this contains a value</th>
<th>*this does not contain a value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs contains a value</td>
<td>calls swap(*(*this), *rhs)</td>
<td>initializes the contained value of *this as if direct-non-list-initializing an object of type T with the expression std::move(*rhs), followed by rhs.val-&gt;T::~T(); postcondition is that *this contains a value and rhs does not contain a value</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>
<td>no effect</td>
</tr>
</tbody>
</table>

Throws: Any exceptions thrown by the operations in the relevant part of Table 39.

Remarks: The expression inside noexcept is equivalent to:

is_nothrow_move_constructible_v<T> && is_nothrow_swappable_v<T>

If any exception is thrown, the results of the expressions bool(*this) and bool(rhs) remain unchanged. If an exception is thrown during the call to function swap, the state of *val and *rhs.val is determined by the exception safety guarantee of swap for lvalues of T. If an exception is thrown during the call to T’s move constructor, the state of *val and *rhs.val is determined by the exception safety guarantee of T’s move constructor.

23.6.3.5 Observers [optional.observe]

constexpr const T* operator->() const;
constexpr T* operator->();

Requires: *this contains a value.
Returns: \( \text{val} \).

Throws: Nothing.

Remarks: These functions shall be constexpr functions.

\[
\begin{align*}
\text{constexpr const } T & \text{& operator*() const&;} \\
& \text{constexpr T & operator*() &;} \\
\end{align*}
\]

Requires: \*\text{this} contains a value.

Returns: \*\text{val}.

Throws: Nothing.

Remarks: These functions shall be constexpr functions.

\[
\begin{align*}
\text{constexpr T & operator*() &;} \\
& \text{constexpr const } T & \text{& operator*() const&;} \\
\end{align*}
\]

Requires: \*\text{this} contains a value.

Effects: Equivalent to: return std::move(*\text{val});

\[
\begin{align*}
\text{constexpr explicit operator bool() const noexcept;} \\
\end{align*}
\]

Returns: true if and only if \*\text{this} contains a value.

Remarks: This function shall be a constexpr function.

\[
\begin{align*}
\text{constexpr bool has_value() const noexcept;} \\
\end{align*}
\]

Returns: true if and only if \*\text{this} contains a value.

Remarks: This function shall be a constexpr function.

\[
\begin{align*}
\text{constexpr const } T & \text{& value() const&;} \\
& \text{constexpr T & value() &;} \\
\end{align*}
\]

Effects: Equivalent to:

\[
\begin{align*}
& \text{return bool(*this) ? *val : throw bad_optional_access();} \\
\end{align*}
\]

\[
\begin{align*}
\text{constexpr T & operator*() &;} \\
& \text{constexpr const } T & \text{& value() const&;} \\
\end{align*}
\]

Effects: Equivalent to:

\[
\begin{align*}
& \text{return bool(*this) ? std::move(*val) : throw bad_optional_access();} \\
\end{align*}
\]

\[
\begin{align*}
\text{template<class } U & \text{> constexpr T value_or(U & v) const&;} \\
\end{align*}
\]

Effects: Equivalent to:

\[
\begin{align*}
& \text{return bool(*this) ? static_cast<T>(std::forward<U>(v)) : throw bad_optional_access();} \\
\end{align*}
\]

Remarks: If is_copy_constructible_v<T &> \& is_convertible_v<&U&, T> is false, the program is ill-formed.

\[
\begin{align*}
\text{template<class } U & \text{> constexpr T value_or(U & v) &;} \\
\end{align*}
\]

Effects: Equivalent to:

\[
\begin{align*}
& \text{return bool(*this) ? std::move(*this) : static_cast<T>(std::forward<U>(v)) ;} \\
\end{align*}
\]

Remarks: If is_move_constructible_v<T &> \& is_convertible_v<&U&, T> is false, the program is ill-formed.

### 23.6.3.6 Modifiers

\[
\text{void reset() noexcept;} \\
\]

Effects: If \*\text{this} contains a value, calls \text{val-}\rightarrow{T::-T()} to destroy the contained value; otherwise no effect.

Postconditions: \*\text{this} does not contain a value.
23.6.4 No-value state indicator

```cpp
struct nullopt_t { see below; }
inline constexpr nullopt_t nullopt(unspecified);
```

1 The struct `nullopt_t` is an empty structure 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.

2 Type `nullopt_t` shall not have a default constructor or an initializer-list constructor, and shall not be an aggregate.

23.6.5 Class `bad_optional_access`

```cpp
class bad_optional_access : public exception {
public:
  bad_optional_access();
};
```

1 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.

```cpp
bad_optional_access();
```

2 Effects: Constructs an object of class `bad_optional_access`.

3 Postconditions: `what()` returns an implementation-defined `ntbs`.

23.6.6 Relational operators

```cpp
template<class T, class U> constexpr bool operator==(const optional<T>& x, const optional<U>& y);
```

1 Requires: The expression `*x == *y` shall be well-formed and its result shall be convertible to `bool`.

   [Note: `T` need not be `EqualityComparable`. —end note]

2 Returns: If `bool(x) != bool(y)`, `false`; otherwise if `bool(x) == false`, `true`; otherwise `*x == *y`.

3 Remarks: Specializations of this function template for which `*x == *y` is a core constant expression shall be `constexpr` functions.

```cpp
template<class T, class U> constexpr bool operator!=(const optional<T>& x, const optional<U>& y);
```

4 Requires: The expression `*x != *y` shall be well-formed and its result shall be convertible to `bool`.

5 Returns: If `bool(x) != bool(y)`, `true`; otherwise, if `bool(x) == false`, `false`; otherwise `*x != *y`.

6 Remarks: Specializations of this function template for which `*x != *y` is a core constant expression shall be `constexpr` functions.

```cpp
template<class T, class U> constexpr bool operator<(const optional<T>& x, const optional<U>& y);
```

7 Requires: The expression `*x < *y` shall be well-formed and its result shall be convertible to `bool`.

8 Returns: If `!y`, `false`; otherwise, if `!x`, `true`; otherwise `*x < *y`.

9 Remarks: Specializations of this function template for which `*x < *y` is a core constant expression shall be `constexpr` functions.

```cpp
template<class T, class U> constexpr bool operator>(const optional<T>& x, const optional<U>& y);
```

10 Requires: The expression `*x > *y` shall be well-formed and its result shall be convertible to `bool`.

11 Returns: If `!x`, `false`; otherwise, if `!y`, `true`; otherwise `*x > *y`.

12 Remarks: Specializations of this function template for which `*x > *y` is a core constant expression shall be `constexpr` functions.

```cpp
template<class T, class U> constexpr bool operator<=(const optional<T>& x, const optional<U>& y);
```

13 Requires: The expression `*x <= *y` shall be well-formed and its result shall be convertible to `bool`.

14 Returns: If `!x`, `true`; otherwise, if `!y`, `false`; otherwise `*x <= *y`.

15 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 \( *x \geq *y \) shall be well-formed and its result shall be convertible to bool.

Returns: If \(!y\), true; otherwise, if \(!x\), false; otherwise \( *x \geq *y \).

Remarks: Specializations of this function template for which \( *x \geq *y \) is a core constant expression shall be constexpr functions.

23.6.7 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;

1 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;

2 Returns: \( \text{bool}(x) \).

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;

3 Returns: \( \text{false} \).

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;

4 Returns: \( \text{true} \).

23.6.8 Comparison with T

[optional.comp_with_t]

template<class T, class U> 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 bool.

[Note: \( T \) need not be EqualityComparable. —end note]

Effects: Equivalent to: return \( \text{bool}(x) \) ? \( *x == v \) : false;

template<class T, class U> 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 bool.

Effects: Equivalent to: return \( \text{bool}(x) \) ? v == \( *x \) : false;

template<class T, class U> constexpr bool operator!=(const optional<T>& x, const U& v);

Requires: The expression \( *x \neq v \) shall be well-formed and its result shall be convertible to bool.

Effects: Equivalent to: return \( \text{bool}(x) \) ? \( *x \neq v \) : true;

template<class T, class U> constexpr bool operator!=(const T& v, const optional<U>& x);

Requires: The expression \( v \neq *x \) shall be well-formed and its result shall be convertible to bool.
Effects: Equivalent to: \( \text{return bool(x) ? v != *x : true; } \)

```
template<class T, class U> 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<class T, class U> 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<class T, class U> 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<class T, class U> 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<class T, class U> 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<class T, class U> 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<class T, class U> 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<class T, class U> 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: \( \text{return bool(x) ? v >= *x : true; } \)

```
23.6.9 Specialized algorithms
```

```
template<class T> void swap(optional<T>& x, optional<T>& y) noexcept(noexcept(x.swap(y)));
```

Effects: Calls \(x\).\(\text{swap}(y)\).

Remarks: This function shall not participate in overload resolution unless \text{is\_move\_constructible\_v<T>} is true and \text{is\_swappable\_v<T>} is true.

```
template<class T> constexpr optional<decay_t<T>> make_optional(T&& v);
```

Returns: \( \text{optional<decay_t<T>>(std::forward<T>(v))} \)

```
template<class T, class... Args> constexpr optional<T> make_optional(Args&&... args);
```

Effects: Equivalent to: \( \text{return optional<T>(in\_place, std::forward<Args>(args)...)} \)

```
template<class T, class U, class... Args> constexpr optional<T> make_optional(initializer_list<U> il, Args&&... args);
```

Effects: Equivalent to: \( \text{return optional<T>(in\_place, il, std::forward<Args>(args)...)} \)

```
23.6.10 Hash support
```

```
template<class T> struct hash<optional<T>>;
```

The specialization \text{hash<optional<T>}> is enabled (23.14.15) if and only if \text{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.

23.7 Variants

23.7.1 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.

23.7.2 Header <variant> synopsis

```cpp
namespace std {

// 23.7.3, class template variant
template<class... Types>
  class variant;

// 23.7.4, 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<const volatile T>;

// not defined

inline constexpr size_t variant_size_v = variant_size<T>::value;

template<class... Types>
  struct variant_size<variant<Types...>>;

// 23.7.5, value access

  constexpr bool holds_alternative(const variant<Types...>&) noexcept;

template<size_t I, class... Types>
  constexpr variant_alternative_t<I, variant<Types...>> get(variant<Types...>&);

template<size_t I, class... Types>
  constexpr variant_alternative_t<I, variant<Types...>>&& get(variant<Types...>&&);

template<size_t I, class... Types>
  constexpr const variant_alternative_t<I, variant<Types...>>& get(const variant<Types...>&);

template<size_t I, class... Types>
  constexpr const variant_alternative_t<I, variant<Types...>>&& get(const variant<Types...>&&);

// 23.7.6, equality

  constexpr bool variant<Types...>::operator==(const variant<Types...>&) const;

  constexpr bool variant<Types...>::operator==(const variant<Types...>&&);  
```

§ 23.7.2
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;

// 23.7.6, 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...>&);

// 23.7.7, visitation
template<class Visitor, class... Variants>
constexpr see below visit(Visitor&&, Variants&&...);

// 23.7.8, class monostate
struct monostate;

// 23.7.9, 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;

// 23.7.10, specialized algorithms
template<class... Types>
void swap(variant<Types...>&, variant<Types...>&) noexcept(see below);

// 23.7.11, class bad_variant_access
class bad_variant_access;

// 23.7.12, hash support
template<class T> struct hash;
template<class... Types> struct hash<variant<Types...>>;
template<> struct hash<monostate>;

} // namespace std

23.7.3 Class template variant

namespace std {
  template<class... Types>
  class variant {
    public:
      // 23.7.3.1, constructors
      constexpr variant() noexcept(see below);
Any instance of `variant` at any given time either holds a value of one of its alternative types, or it 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...`. It is implementation-defined whether over-aligned types are supported.

All types in `Types...` shall be (possibly cv-qualified) object types that are not arrays.

A program that instantiates the definition of `variant` with no template arguments is ill-formed.

### 23.7.3.1 Constructors

In the descriptions that follow, let `i` be in the range `[0, sizeof...(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`.

**Postconditions:** `valueless_by_exception()` is `false` and `index()` is `0`.

**Throws:** Any exception thrown by the value-initialization of `T_0`.
5 Remarks: This function shall be constexpr if and only if the value-initialization of the alternative type T₀ would satisfy the requirements for a constexpr function. The expression inside noexcept is equivalent to is_nothrow_default_constructible_v<variant<T₀>>. This function shall not participate in overload resolution unless is_default_constructible_v<variant<T₀>> is true. [Note: See also class monostate. —end note]

variant(const variant& w);
6 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.
7 Throws: Any exception thrown by direct-initializing any Tᵢ for all i.
8 Remarks: This constructor shall be defined as deleted unless is_default_constructible_v<Tᵢ> is true for all i.

variant(variant&& w) noexcept(see below);
9 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.
10 Throws: Any exception thrown by move-constructing any Tᵢ for all i.
11 Remarks: The expression inside noexcept is equivalent to the logical AND of is_nothrow_move_constructible_v<Tᵢ> for all i. This function shall not participate in overload resolution unless is_move_constructible_v<variant<Tᵢ>> is true for all i.

template<class T> constexpr variant(T&& t) noexcept(see below);
12 Let Tᵢ be a type that is determined as follows: build an imaginary function FUN(Tᵢ) for each alternative type Tᵢ. The overload FUN(Tᵢ) selected by overload resolution for the expression FUN(std::forward<T>(t)) defines the alternative Tᵢ which is the type of the contained value after construction.
13 Effects: Initializes *this to hold the alternative type Tᵢ and direct-initializes the contained value as if direct-non-list-initializing it with std::forward<T>(t).
14 Postconditions: holds_alternative<Tᵢ>(*this) is true.
15 Throws: Any exception thrown by the initialization of the selected alternative Tᵢ.

variant(variant&& w) noexcept(see below);
19 Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(args)....
20 Postconditions: holds_alternative<T*>(*this) is true.
21 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, Args...> 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... Args>
constexpr explicit variant(in_place_type_t<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)....

**Postconditions:** 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>&, Args...> is true. If T's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```cpp
template<size_t I, class... Args> constexpr explicit variant(in_place_index_t<I>, Args&&... args);
```

**Effects:** Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(args)....

**Postconditions:** index() is I.

**Throws:** Any exception thrown by calling the selected constructor of T.

**Remarks:** This function shall not participate in overload resolution unless

1. I is less than sizeof...(Types)
2. is_constructible_v<T, Args...> is true.

If T's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

```cpp
template<size_t I, class U, class... Args>
constexpr explicit variant(in_place_index_t<I>, 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)....

**Postconditions:** index() is I.

**Remarks:** This function shall not participate in overload resolution unless

1. I is less than sizeof...(Types)
2. is_constructible_v<T, initializer_list<U>&, Args...> is true.

If T's selected constructor is a constexpr constructor, this constructor shall be a constexpr constructor.

### 23.7.3.2 Destructor

-variant();

**Effects:** If valueless_by_exception() is false, destroys the currently contained value.

**Remarks:** If is_trivially_destructible_v<T> == true for all T, then this destructor shall be a trivial destructor.

### 23.7.3.3 Assignment

```cpp
variant& operator=(const variant& rhs);
```

**Let j be rhs.index().**

**Effects:**

1. If neither *this nor rhs holds a value, there is no effect.
2. Otherwise, if *this holds a value but rhs does not, destroys the value contained in *this and sets *this to not hold a value.
3. Otherwise, if index() == j, assigns the value contained in rhs to the value contained in *this.
4. Otherwise, if either is_nothrow_copy_constructible_v<T> is true or is_nothrow_move_constructible_v<T> is false, equivalent to emplace<j>(get<j>(rhs)).
(2.5) Otherwise, equivalent to \texttt{operator=(variant(rhs))}.

\begin{itemize}
\item \textbf{Returns: }*this.
\item \textbf{Postconditions: }\texttt{index() == rhs.index()}. \end{itemize}

\textbf{Remarks: }This operator shall be defined as deleted unless \texttt{is\_copy\_constructible\_v<T_i>} \&\& \texttt{is\_copy\_assignable\_v<T_i>} is true for all \texttt{i}.

\texttt{variant\& operator=(variant\&\& rhs) noexcept(see below);}

\begin{itemize}
\item Let \texttt{j be rhs.index().}
\end{itemize}

\item \textbf{Effects:}
\begin{itemize}
\item If neither \texttt{*this} nor \texttt{rhs} holds a value, there is no effect.
\item Otherwise, if \texttt{*this} holds a value but \texttt{rhs} does not, destroys the value contained in \texttt{*this} and sets \texttt{*this} to not hold a value.
\item Otherwise, if \texttt{index() == j}, assigns \texttt{get\langle j\rangle(std::move(rhs))} to the value contained in \texttt{*this}.
\item Otherwise, equivalent to \texttt{emplace\langle j\rangle(get\langle j\rangle(std::move(rhs)))}.
\end{itemize}

\item \textbf{Returns: }\texttt{*this}.

\item \textbf{Remarks: }This function shall not participate in overload resolution unless \texttt{is\_move\_constructible\_v<T_i>} \&\& \texttt{is\_move\_assignable\_v<T_i>} is true for all \texttt{i}. The expression inside \texttt{noexcept} is equivalent to: \texttt{is\_nothrow\_move\_constructible\_v<T_i>} \&\& \texttt{is\_nothrow\_move\_assignable\_v<T_i>} for all \texttt{i}.

\item \textbf{Remarks: }This function shall not participate in overload resolution unless \texttt{is\_move\_constructible\_v<T_i>} \&\& \texttt{is\_move\_assignable\_v<T_i>} is true for all \texttt{i}. The expression inside \texttt{noexcept} is equivalent to: \texttt{is\_nothrow\_move\_constructible\_v<T_i>} \&\& \texttt{is\_nothrow\_move\_assignable\_v<T_i>} for all \texttt{i}.

\item \textbf{Effects:}
\begin{itemize}
\item If \texttt{*this} holds a \texttt{T_j}, assigns \texttt{std::forward\langle T\rangle(t)} to the value contained in \texttt{*this}.
\item Otherwise, if \texttt{is\_nothrow\_constructible\_v<T_j, T> || !is\_nothrow\_move\_constructible\_v<T_i>} is true, equivalent to \texttt{emplace\langle j\rangle(std::forward\langle T\rangle(t))}.
\item Otherwise, equivalent to \texttt{operator=(variant(std::forward\langle T\rangle(t)))}.
\end{itemize}

\item \textbf{Postconditions: }\texttt{holds\_alternative\_T_j\langle*this\rangle} is true, with \texttt{T_j selected by the imaginary function overload resolution described above.}

\item \textbf{Returns: }\texttt{*this}.

\item \textbf{Remarks: }This function shall not participate in overload resolution unless
\begin{itemize}
\item \texttt{is\_same\_v<remove\_cvref\_t\langle T\rangle, variant> is false},
\item \texttt{is\_assignable\_v\langle T_i, T\rangle, T\rangle \&\& \texttt{is\_constructible\_v\langle T_j, T\rangle} is true, and
\item the expression \texttt{FUN(std::forward\langle T\rangle(t))} (with \texttt{FUN} being the above-mentioned set of imaginary functions) is well-formed.
\end{itemize}

\texttt{[Note: variant\_string, string\rangle v;}
\texttt{v = "abc";}

\texttt{is ill-formed, as both alternative types have an equally viable constructor for the argument. — end note]}

\item The expression inside \texttt{noexcept} is equivalent to:
\texttt{is\_nothrow\_assignable\_v\langle T_i, T\rangle \&\& is\_nothrow\_constructible\_v\langle T_j, T\rangle}
— If an exception is thrown during the assignment of `std::forward<T>(t)` to the value contained in `*this`, the state of the contained value and `t` are as defined by the exception safety guarantee of the assignment expression; `valueless_by_exception()` will be `false`.

— If an exception is thrown during the initialization of the contained value, the `variant` object might not hold a value.

### 23.7.3.4 Modifiers

```cpp
template<class T, class... Args> T& emplace(Args&&... args);
```

- Let `I` be the zero-based index of `T` in `Types`.
- **Effects:** Equivalent to: `return emplace<I>(std::forward<Args>(args)...);`
- **Remarks:** This function shall not participate in overload resolution unless `is_constructible_v<T, Args...>` is true, and `T` occurs exactly once in `Types`.

```cpp
template<class T, class U, class... Args> T& emplace(initializer_list<U> il, Args&&... args);
```

- Let `I` be the zero-based index of `T` in `Types`.
- **Effects:** Equivalent to: `return emplace<I>(il, std::forward<Args>(args)...);`
- **Remarks:** This function shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args...>` is true, and `T` occurs exactly once in `Types`.

```cpp
template<size_t I, class... Args> variant_alternative_t<I, variant<Types...>>& emplace(Args&&... args);
```

- **Requires:** `I < sizeof...(Types)`.
- **Effects:** Destroys the currently contained value if `valueless_by_exception()` is `false`. Then initializes the contained value as if direct-non-list-initializing a value of type `T_I` with the arguments `std::forward<Args>(args)...`.
- **Postconditions:** `index()` is `I`.
- **Returns:** A reference to the new contained value.
- **Throws:** Any exception thrown during the initialization of the contained value.
- **Remarks:** This function shall not participate in overload resolution unless `is_constructible_v<T_I, Args...>` is true. If an exception is thrown during the initialization of the contained value, the `variant` might not hold a value.

```cpp
template<size_t I, class U, class... Args> variant_alternative_t<I, variant<Types...>>& emplace(initializer_list<U> il, Args&&... args);
```

- **Requires:** `I < sizeof...(Types)`.
- **Effects:** Destroys the currently contained value if `valueless_by_exception()` is `false`. Then initializes the contained value as if direct-non-list-initializing a value of type `T_I` with the arguments `il, std::forward<Args>(args)...`.
- **Postconditions:** `index()` is `I`.
- **Returns:** A reference to the new contained value.
- **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.

### 23.7.3.5 Value status

```cpp
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

---

\[ § 23.7.3.5 \]
struct S { operator int() { throw 42; }};

variant<float, int> v{12.f};
v.emplace<1>(S());

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.

23.7.3.6 Swap

void swap(variant& rhs) noexcept(see below);

Requires: Lvalues of type Tᵢ shall be swappable (20.5.3.2) and is_move_constructible_v<Tᵢ> shall be true for all i.

Effects:

1. If valueless_by_exception() && rhs.valueless_by_exception() no effect.
2. Otherwise, if index() == rhs.index(), calls swap(get<i>(*this), get<i>(rhs)) where i is index().
3. Otherwise, exchanges values of rhs and *this.

Throws: If index() == rhs.index(), any exception thrown by swap(get<i>(*this), get<i>(rhs)) with i being index(). Otherwise, any exception thrown by the move constructor of Tᵢ or Tⱼ 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 Tᵢ 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<Tᵢ> && is_nothrow_swappable_v<Tᵢ> for all i.

23.7.4 variant helper classes

template<class T> struct variant_size;

Remarks: All specializations of variant_size shall meet the UnaryTypeTrait requirements (23.15.1) 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 variant_size<T> of the cv-unqualified type T. Then each of the three templates shall meet the UnaryTypeTrait requirements (23.15.1) with a base characteristic of integral_constant<size_t, VS::value>.

template<class... Types>
struct variant_size<variant<Types...>> : integral_constant<size_t, sizeof...(Types)> { };

template<size_t I, class T> class variant_alternative<I, const T>;
template<size_t I, class T> class variant_alternative<I, volatile T>;
template<size_t I, class T> class variant_alternative<I, const volatile T>;

Let VA denote variant_alternative<I, T> of the cv-unqualified type T. Then each of the three templates shall meet the TransformationTrait requirements (23.15.1) with a member typedef type that names the following type:

1. For the first specialization, add_const_t<VA::type>,
2. for the second specialization, add_volatile_t<VA::type>, and
3. for the third specialization, add_cv_t<VA::type>.

§ 23.7.4
variant_alternative<I, variant<Types...>::type

Requires: I < sizeof...(Types). The program is ill-formed if I is out of bounds.
Value: The type T_I.

23.7.5 Value access

template<class T, class... Types>
constexpr bool holds_alternative(const variant<Types...>& v) noexcept;

Requires: The type T occurs exactly once in Types... Otherwise, the program is ill-formed.
Returns: true if index() is equal to the zero-based index of T in Types....

Effects: If v.index() is I, returns a reference to the object stored in the variant. Otherwise, throws an exception of type bad_variant_access.

Requires: I < sizeof...(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 bad_variant_access.

3
Requires: I < sizeof...(Types). Otherwise the program is ill-formed.
Returns: A pointer to the value stored in the variant, if v != nullptr and v->index() == I. Otherwise, returns nullptr.

23.7.6 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: \( \text{get}\langle i\rangle(v) \neq \text{get}\langle i\rangle(w) \) is a valid expression returning a type that is convertible to bool, for all \( i \).

Returns: If \( v.\text{index}() \neq w.\text{index}() \), true; otherwise if \( v.\text{valueless\_by\_exception}() \), false; otherwise \( \text{get}\langle i\rangle(v) \neq \text{get}\langle i\rangle(w) \) with \( i \) being \( v.\text{index}() \).

template<class... Types>
constexpr bool operator<(const variant<Types...>& v, const variant<Types...>& w);

Requires: \( \text{get}\langle i\rangle(v) < \text{get}\langle i\rangle(w) \) is a valid expression returning a type that is convertible to bool, for all \( i \).

Returns: If \( w.\text{valueless\_by\_exception}() \), false; otherwise if \( v.\text{valueless\_by\_exception}() \), true; otherwise if \( v.\text{index}() < w.\text{index}() \), true; otherwise if \( v.\text{index}() > w.\text{index}() \), false; otherwise \( \text{get}\langle i\rangle(v) < \text{get}\langle i\rangle(w) \) with \( i \) being \( v.\text{index}() \).

template<class... Types>
constexpr bool operator>(const variant<Types...>& v, const variant<Types...>& w);

Requires: \( \text{get}\langle i\rangle(v) > \text{get}\langle i\rangle(w) \) is a valid expression returning a type that is convertible to bool, for all \( i \).

Returns: If \( v.\text{valueless\_by\_exception}() \), false; otherwise if \( w.\text{valueless\_by\_exception}() \), true; otherwise if \( v.\text{index}() > w.\text{index}() \), true; otherwise if \( v.\text{index}() < w.\text{index}() \), false; otherwise \( \text{get}\langle i\rangle(v) > \text{get}\langle i\rangle(w) \) with \( i \) being \( v.\text{index}() \).

template<class... Types>
constexpr bool operator<=(const variant<Types...>& v, const variant<Types...>& w);

Requires: \( \text{get}\langle i\rangle(v) \leq \text{get}\langle i\rangle(w) \) is a valid expression returning a type that is convertible to bool, for all \( i \).

Returns: If \( v.\text{valueless\_by\_exception}() \), true; otherwise if \( w.\text{valueless\_by\_exception}() \), false; otherwise if \( v.\text{index}() < w.\text{index}() \), true; otherwise if \( v.\text{index}() > w.\text{index}() \), false; otherwise \( \text{get}\langle i\rangle(v) \leq \text{get}\langle i\rangle(w) \) with \( i \) being \( v.\text{index}() \).

template<class... Types>
constexpr bool operator>=(const variant<Types...>& v, const variant<Types...>& w);

Requires: \( \text{get}\langle i\rangle(v) \geq \text{get}\langle i\rangle(w) \) is a valid expression returning a type that is convertible to bool, for all \( i \).

Returns: If \( w.\text{valueless\_by\_exception}() \), true; otherwise if \( v.\text{valueless\_by\_exception}() \), false; otherwise if \( v.\text{index}() > w.\text{index}() \), true; otherwise if \( v.\text{index}() < w.\text{index}() \), false; otherwise \( \text{get}\langle i\rangle(v) \geq \text{get}\langle i\rangle(w) \) with \( i \) being \( v.\text{index}() \).

23.7.7 Visitation

template<class Visitor, class... Variants>
constexpr see below visit(Visitor&& vis, Variants&&... vars);

Requires: The expression in the Effects: element shall be a valid expression of the same type and value category, for all combinations of alternative types of all variants. Otherwise, the program is ill-formed.

Effects: Let is... be vars.index()... Then returns \( \text{invoke}(\text{forward}\langle\text{Visitor}\rangle(vis), \text{get}\langle\text{is}\rangle(\text{forward}\langle\text{Variants}\rangle(vars))...); \) (23.14.3).

Remarks: The return type is the common type of all possible \( \text{invoke} \) expressions of the Effects: element.

Throws: bad\_variant\_access if any variant in vars is valueless\_by\_exception().

Complexity: For sizeof...(Variants) \( \leq 1 \), the invocation of the callable object is implemented in constant time, i.e., it does not depend on sizeof...(Types). For sizeof...(Variants) \( > 1 \), the invocation of the callable object has no complexity requirements.
23.7.8 Class monostate

```cpp
struct monostate {};
```

The class `monostate` can serve as a first alternative type for a `variant` to make the `variant` type default constructible.

23.7.9 monostate relational operators

```cpp
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; }
constexpr bool operator==(monostate, monostate) noexcept { return true; }
constexpr bool operator!=(monostate, monostate) noexcept { return false; }
```

1 [Note: `monostate` objects have only a single state; they thus always compare equal. — end note]

23.7.10 Specialized algorithms

```cpp
template<class... Types>
void swap(variant<Types...>& v, variant<Types...>& w) noexcept(see below);
```

1 Effects: Equivalent to `v.swap(w)`.
2 Remarks: This function shall not participate in overload resolution unless `is_move_constructible_v<T_i> && is_swappable_v<T_i>` is `true` for all `i`. The expression inside `noexcept` is equivalent to `noexcept(v.swap(w))`.

23.7.11 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 `bad_variant_access` are thrown to report invalid accesses to the value of a `variant` object.

```cpp
bad_variant_access() noexcept;

const char* what() const noexcept override;
```

1 Constructs a `bad_variant_access` object.
2 Returns: An implementation-defined NTBS.

23.7.12 Hash support

```cpp
template<class... Types> struct hash<variant<Types...>>;
```

1 The specialization `hash<variant<Types...>>` is enabled (23.14.15) if and only if every specialization in `hash<remove_const_t<Types>>...` is enabled. The member functions are not guaranteed to be `noexcept`.

```cpp
template<> struct hash<monostate>;
```

1 The specialization is enabled (23.14.15).

23.8 Storage for any type

This subclause describes components that C++ programs may use to perform operations on objects of a discriminated type.

1 [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]
23.8.1 Header <any> synopsis

namespace std {
  // 23.8.2, class bad_any_cast
  class bad_any_cast;

  // 23.8.3, class any
  class any;

  // 23.8.4, non-member functions
  void swap(any& x, any& y) noexcept;
  template<class T, class... Args>
  any make_any(Args&& ...args);
  template<class T, class U, class... Args>
  any make_any(initializer_list<U> il, Args&& ...args);
  template<class T>
  T any_cast(const any& operand);
  template<class T>
  T any_cast(any& operand);
  template<class T>
  T any_cast(any&& operand);
  template<class T>
  const T* any_cast(const any* operand) noexcept;
  template<class T>
  T* any_cast(any* operand) noexcept;
}

23.8.2 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 (23.8.4).

const char* what() const noexcept override;

2 Returns: An implementation-defined ntbs.

3 Remarks: The message may be a null-terminated multibyte string (20.4.2.1.5.2), suitable for conversion and display as a wstring (24.3, 25.4.1.4).

23.8.3 Class any

class any {
public:
  // 23.8.3.1, construction and destruction
  constexpr any() noexcept;
  any(const any& other);
  any(any&& other) noexcept;
  ~any();

  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&&...);

  // 23.8.3.2, assignments
  any& operator=(const any& rhs);
  any& operator=(any&& rhs) noexcept;
};
template<class T> any& operator=(T&& rhs);

// 23.8.3.3, 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;

// 23.8.3.4, 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 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. [Example: where the object constructed is holding only an int. —end example] Such small-object optimization shall only be applied to types T for which is_nothrow_move_constructible_v<T> is true.

23.8.3.1 Construction and destruction [any.cons]

cconstexpr any() noexcept;

1 Postconditions: has_value() is false.

any(const any& other);

2 Effects: If other.has_value() is false, constructs an object that has no value. Otherwise, equivalent to any(in_place_type<T>, any_cast<const T&>(other)) where T is the type of the contained value.

3 Throws: Any exceptions arising from calling the selected constructor for the contained value.

any(any&& other) noexcept;

4 Effects: If other.has_value() is false, constructs an object that has no value. Otherwise, constructs an object of type any that contains either the contained value of other, or contains an object of the same type constructed from the contained value of other considering that contained value as an rvalue.

5 Postconditions: other is left in a valid but otherwise unspecified state.

template<class T>
    any(T&& value);

1 Let VT be decay_t<T>.

2 Requires: VT shall satisfy the CopyConstructible requirements.

3 Effects: Constructs an object of type any that contains an object of type VT direct-initialized with std::forward<T>(value).

4 Remarks: This constructor shall not participate in overload resolution unless VT is not the same type as any, VT is not a specialization of in_place_type_t, and is_copy_constructible_v<VT> is true.

5 Throws: Any exception thrown by the selected constructor of VT.

template<class T, class... Args>
    explicit any(in_place_type_t_t<T>, Args&&... args);

1 Let VT be decay_t<T>.

2 Requires: VT shall satisfy the CopyConstructible requirements.

3 Effects: Initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments std::forward<Args>(args)....

4 Postconditions: *this contains a value of type VT.
Throws: Any exception thrown by the selected constructor of VT.

Remarks: This constructor 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>
explicit any(in_place_type_t<T>, initializer_list<U> il, Args&&... args);
```

Let `VT` be `decay_t<T>`.

Requires: VT shall satisfy the CopyConstructible requirements.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments `il`, `std::forward<Args>(args)....`

Postconditions: *this contains a value.

Throws: Any exception thrown by the selected constructor of VT.

Remarks: This constructor 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.

~any();

Effects: As if by `reset()`.

### 23.8.3.2 Assignment

```cpp
any& operator=(const any& rhs);
```

Effects: As if by `any(rhs).swap(*this)`. No effects if an exception is thrown.

Returns: *this.

Throws: Any exceptions arising from the copy constructor for the contained value.

```cpp
any& operator=(any&& rhs) noexcept;
```

Effects: As if by `any(std::move(rhs)).swap(*this)`.

Returns: *this.

Postconditions: The state of *this is equivalent to the original state of rhs 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 CopyConstructible 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.

### 23.8.3.3 Modifiers

```cpp
template<class T, class... Args>
decay_t<T>& emplace(Args&&... args);
```

Let `VT` be `decay_t<T>`.

 Requires: VT shall satisfy the CopyConstructible requirements.

Effects: Calls `reset()`. Then initializes the contained value as if direct-non-list-initializing an object of type VT with the arguments `std::forward<Args>(args)....`

Postconditions: *this contains a value.

Returns: A reference to the new contained value.
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 CopyConstructible requirements.
  
  Effects: Calls reset(). Then initializes the contained value as if direct-non-list-initializing an object of
  type VT with the arguments il, std::forward<Args>(args)...
  
  Postconditions: *this contains a value.
  
  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.

void reset() noexcept;
  
  Effects: If has_value() is true, destroys the contained value.
  
  Postconditions: has_value() is false.

void swap(any& rhs) noexcept;
  
  Effects: Exchanges the states of *this and rhs.

23.8.3.4 Observers

bool has_value() const noexcept;
  
  Returns: true if *this contains an object, otherwise false.

const type_info& type() const noexcept;
  
  Returns: typeid(T) if *this has a contained value of type T, otherwise typeid(void).
  
  [Note: Useful for querying against types known either at compile time or only at runtime. — end
  note]

23.8.4 Non-member functions

void swap(any& x, any& y) noexcept;
  
  Effects: As if by x.swap(y).

template<class T, class... Args>
  any make_any(Args&&... args);
  
  Effects: Equivalent to: return any(in_place_type<T>, std::forward<Args>(args)...);

template<class T, class U, class... Args>
  any make_any(initializer_list<U> il, Args&&... args);
  
  Effects: Equivalent to: return any(in_place_type<T>, il, std::forward<Args>(args)...);

  template<class T>
  T any_cast(const any& operand);
  template<class T>
  T any_cast(any& operand);
  template<class T>
  T any_cast(any&& operand);
  
  Let U be the type remove_cvref_t<T>.
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.

Returns: For the first and second overload, static_cast<T>(*any_cast<U>(&operand)). For the third overload, static_cast<T>(std::move(*any_cast<U>(&operand)));

Throws: bad_any_cast if operand.type() != typeid(remove_reference_t<T>);

Example:
```
any x(5);
assert(any_cast<int>(x) == 5); // cast to value
any_cast<int&>(x) = 10; // cast to reference
assert(any_cast<int>(x) == 10);

x = "Meow"; // x holds const char*
assert(strcmp(any_cast<const char*>(x), "Meow") == 0);
any_cast<const char&>(x) = "Harry";
assert(strcmp(any_cast<const char*>(x), "Harry") == 0);

x = string("Meow"); // x holds string
string s, s2("Jane");
s = move(any_cast<string&>(x)); // move from any
assert(s == "Meow");
any_cast<string&>(x) = move(s2); // move to any
assert(any_cast<const string&>(x) == "Jane");

string cat("Meow");
const any y(cat); // const y holds string
assert(any_cast<const string&>(y) == cat);
any_cast<string&>(y); // error; cannot // any_cast away const

—end example
```

```
template<class T>
const T* any_cast(const any* operand) noexcept;
template<class T>
T* any_cast(any* operand) noexcept;

Returns: If operand != nullptr && 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
```

23.9 Bitsets

23.9.1 Header <bitset> synopsis

```
#include <string>
#include <iosfwd> // for istream (30.7.1), ostream (30.7.2), see 30.3.1

namespace std {

template<size_t N> class bitset;

// 23.9.4, 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;
```

§ 23.9.1
The header `<bitset>` defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

### 23.9.2 Class template `bitset`

```cpp
namespace std {

    // bit reference
    class reference {
        friend class bitset;
        reference() noexcept;
        ~reference() noexcept;
        reference& operator=(bool x) noexcept;
        bool operator~() const noexcept;
        operator bool() const noexcept;
        reference& flip() noexcept;
    };

    // 23.9.2.1, constructors
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;
    template<class charT, class traits, class Allocator>
    explicit bitset(
        const basic_string<charT, traits, Allocator>& str,
        typename basic_string<charT, traits, Allocator>::size_type pos = 0,
        typename basic_string<charT, traits, Allocator>::size_type n = basic_string<charT, traits, Allocator>::npos,
        charT zero = charT('0'),
        charT one = charT('1'));
    template<class charT>
    explicit bitset(
        const charT* str,
        typename basic_string<charT>::size_type n = basic_string<charT>::npos,
        charT zero = charT('0'),
        charT one = charT('1'));

    // 23.9.2.2, 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];

} // namespace std
```

§ 23.9.2
The class template `bitset<N>` describes an object that can store a sequence consisting of a fixed number of bits, \(N\).

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 \(p\). When converting between an object of class `bitset<N>` and a value of some integral type, bit position \(p\) corresponds to the bit value \(1 \ll p\).

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` (22.2.4);
2. **An out-of-range error** is associated with exceptions of type `out_of_range` (22.2.6);
3. **An overflow error** is associated with exceptions of type `overflow_error` (22.2.9).

### 23.9.2.1 bitset constructors [bitset.cons]

```cpp
constexpr bitset() noexcept;

Effects: Constructs an object of class `bitset<N>`, initializing all bits to zero.
```

```cpp
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 \(\text{val}\). \(M\) 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.

```cpp
template<class charT, class traits, class Allocator>
explicit bitset(
    const basic_string<charT, traits, Allocator>& str,
    typename basic_string<charT, traits, Allocator>::size_type pos = 0,
    typename basic_string<charT, traits, Allocator>::size_type n = basic_string<charT, traits, Allocator>::npos,
    charT zero = charT('0'),
    charT one = charT('1'))
```

**Throws:** `out_of_range` if \(\text{pos} > \text{str.size()}\) or `invalid_argument` if an invalid character is found (see below).

**Effects:** Determines the effective length \(\text{rlen}\) of the initializing string as the smaller of \(n\) and \(\text{str.size()} - \text{pos}\).
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.

```cpp
template<class charT>
explicit bitset(const charT* str,
    typename basic_string<charT>::size_type n = basic_string<charT>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

`Effects:` Constructs an object of class `bitset<N>` as if by:

```cpp
bitset(n == basic_string<charT>::npos ? basic_string<charT>(str) :
    basic_string<charT>(str, n),
0, n, zero, one)
```

### 23.9.2.2 bitset members

- `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`.

- `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`.

- `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`.

- `operator<<=(size_t pos) noexcept;`  
  
  `Effects:` Replaces each bit at position `I` in `*this` with a value determined as follows:

  (7.1) If `I < pos`, the new value is zero;
  
  (7.2) If `I >= pos`, the new value is the previous value of the bit at position `I - pos`.
  
  `Returns:` `*this`.

- `operator>>=(size_t pos) noexcept;`  
  
  `Effects:` Replaces each bit at position `I` in `*this` with a value determined as follows:

  (9.1) If `pos >= N - I`, the new value is zero;
  
  (9.2) If `pos < N - I`, the new value is the previous value of the bit at position `I + pos`.
  
  `Returns:` `*this`.

- `set() noexcept;`  
  
  `Effects:` Sets all bits in `*this`.
  
  `Returns:` `*this`.

§ 23.9.2.2
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();

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();

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 zero, bit value one becomes the character one.
Returns: The created object.

size_t count() const noexcept;

Returns: A count of the number of bits set in *this.
constexpr size_t size() const noexcept;

Returns: N.

bool operator==(const bitset<N>& rhs) const noexcept;

Returns: true if the value of each bit in *this equals the value of the corresponding bit in rhs.

bool 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: count() == size().

bool any() const noexcept;

Returns: count() != 0.

bool none() const noexcept;

Returns: count() == 0.

bitset<N> operator<<=(size_t pos) const noexcept;

Returns: bitset<N>(*this) <<= pos.

bitset<N> operator>>=(size_t pos) const noexcept;

Returns: bitset<N>(*this) >>= pos.

constexpr bool operator[] (size_t pos) const;

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.

23.9.3 bitset hash support

template<size_t N> struct hash<bitset<N>>;

The specialization is enabled (23.14.15).

23.9.4 bitset operators

bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) &= rhs.

bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) |= rhs.

bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs) noexcept;

Returns: bitset<N>(lhs) ^= rhs.

§ 23.9.4
template<class charT, class traits, size_t N>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, bitset<N>& x);

A formatted input function (30.7.4.2).

Effects: Extracts up to \(N\) characters from is. Stores these characters in a temporary object \(str\) of type basic_string<charT, traits>, then evaluates the expression \(x = \text{bitset}<N>(str)\). Characters are extracted and stored until any of the following occurs:

- \(N\) characters have been extracted and stored;
- end-of-file occurs on the input sequence;
- the next input character is neither \(\text{is.widen('0')}\) nor \(\text{is.widen('1')}\) (in which case the input character is not extracted).

If no characters are stored in \(str\), calls \(\text{is.setstate(ios_base::failbit)}\) (which may throw ios_base::failure (30.5.5.4)).

Returns: is.

template<class charT, class traits, size_t N>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);

Returns:
\[\text{os} \ll x\text{.template to_string<charT, traits, allocator<charT>>(
\text{use_facet<ctype<charT>>(os.getloc()).widen('0'),
\text{use_facet<ctype<charT>>(os.getloc()).widen('1')})
\text{(see 30.7.5.2).}}\]

23.10 Memory [memory]

23.10.1 In general [memory.general]

This subclause describes the contents of the header \textless memory\textgreater (23.10.2) and some of the contents of the header \textless cstdlib\textgreater (21.2.2).

23.10.2 Header \textless memory\textgreater synopsis [memory.syn]

The header \textless memory\textgreater 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 (23.10.3–23.10.11). The header also defines the templates unique_ptr, shared_ptr, weak_ptr, and various function templates that operate on objects of these types (23.11).

namespace std {

    // 23.10.3, pointer traits
    template<class Ptr> struct pointer_traits;
    template<class T> struct pointer_traits<T*>;

    // 23.10.4, pointer conversion
    template<class Ptr>
    auto to_address(const Ptr& p) noexcept;
    template<class T>
    constexpr T* to_address(T* p) noexcept;

    // 23.10.5, 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;

    // 23.10.6, pointer alignment function
    void* align(size_t alignment, size_t size, void* ptr, size_t& space);
}

§ 23.10.2
// 23.10.7, allocator argument tag
struct allocator_arg_t { explicit allocator_arg_t() = default; 
};
inline constexpr allocator_arg_t allocator_arg{ };  

// 23.10.8, uses_allocator
template<class T, class Alloc> struct uses_allocator;

// 23.10.9, allocator traits
template<class Alloc> struct allocator_traits;

// 23.10.10, the default allocator
template<class T> class allocator;
template<class T, class U>
bool operator==(const allocator<T>&, const allocator<U>&) noexcept;

// 23.10.11, specialized algorithms
template<class T>
constexpr T* addressof(T& r) noexcept;
template<class T>
const T* addressof(const T&&) = delete;

// see 28.4.5
template<class ForwardIterator>
void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
void uninitialized_default_construct(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Size>
ForwardIterator uninitialized_default_construct_n(ForwardIterator first, Size n);

template<class ExecutionPolicy, class ForwardIterator, class Size>
ForwardIterator uninitialized_default_construct_n(ExecutionPolicy&& exec, ForwardIterator first, Size n);

template<class ForwardIterator>
void uninitialized_value_construct(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
void uninitialized_value_construct(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Size>
ForwardIterator uninitialized_value_construct_n(ForwardIterator first, Size n);

template<class ExecutionPolicy, class ForwardIterator, class Size>
ForwardIterator uninitialized_value_construct_n(ExecutionPolicy&& exec, ForwardIterator first, Size n);

template<class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
ForwardIterator result);

template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(ExecutionPolicy&& exec, InputIterator first, InputIterator last,
ForwardIterator result);

template<class InputIterator, class Size, class ForwardIterator>
ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
ForwardIterator result);

template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
ForwardIterator uninitialized_copy_n(ExecutionPolicy&& exec, InputIterator first, Size n,
ForwardIterator result);

template<class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
ForwardIterator result);

template<class ExecutionPolicy, class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_move(ExecutionPolicy&& exec, InputIterator first, InputIterator last,
ForwardIterator result);
template<class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator> uninitialized_move_n(InputIterator first, Size n,
ForwardIterator result);

template<class ExecutionPolicy, class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator> uninitialized_move_n(ExecutionPolicy&& exec, // see 28.4.5
InputIterator first, Size n, ForwardIterator result);

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 28.4.5
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 28.4.5
ForwardIterator first, Size 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 28.4.5
ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Size>
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n);

template<class ExecutionPolicy, class ForwardIterator, class Size>
ForwardIterator uninitialized_fill_n(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, Size n);

// 23.11.1, class template unique_ptr

template<class T> struct default_delete;

template<class T> struct default_delete<T[]>; // T is not array

// T is U[]
template<class T, class D = default_delete<T[]>>
class unique_ptr;

// T is U[N]
template<class T, class... Args>
class unique_ptr<T>
make_unique(Args&&... args);

template<class T>
class unique_ptr<T>
make_unique(size_t n);

// unspecified make_unique(Args&&...) = delete;

template<class T, class D>
void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y) noexcept;

bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);

bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;

bool operator==(nullptr_t, const unique_ptr<T, D>& y) noexcept;

§ 23.10.2
template<class T, class D>
  bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;

template<class T, class D>
  bool operator!=(nullptr_t, const unique_ptr<T, D>& y) noexcept;

template<class T, class D>
  bool operator<(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
  bool operator<(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
  bool operator<=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
  bool operator<=(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
  bool operator>(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
  bool operator>(nullptr_t, const unique_ptr<T, D>& y);

template<class T, class D>
  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 E, class T, class Y, class D>
  basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os, const unique_ptr<Y, D>& p);

// 23.11.2, class bad_weak_ptr
class bad_weak_ptr;

// 23.11.3, class template shared_ptr
template<class T> class shared_ptr;

// 23.11.3.6, shared_ptr creation
template<class T, class... Args>
  shared_ptr<T> make_shared(Args&&... args); // T is not array

template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args); // T is not array

template<class T>
  shared_ptr<T> make_shared(size_t N); // T is U[]

template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a, size_t N); // T is U[]

template<class T>
  shared_ptr<T> make_shared(); // T is U[N]

template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a); // T is U[N]

template<class T>
  shared_ptr<T> make_shared(size_t N, const remove_extent_t<T>& u); // T is U[]

template<class T, class A>
  shared_ptr<T> allocate_shared(const A& a, size_t N,
                              const remove_extent_t<T>& u); // T is U[]

template<class T>
  shared_ptr<T> 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]

// 23.11.3.7, 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, 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;

// 23.11.3.8, shared_ptr specialized algorithms
template<class T>
  void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 23.11.3.9, 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> dynamic_pointer_cast(const 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> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

// 23.11.3.10, shared_ptr get_deleter
template<class D, class T>
  D* get_deleter(const shared_ptr<T>& p) noexcept;

// 23.11.3.11, 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);

// 23.11.4, class template weak_ptr
template<class T> class weak_ptr;

// 23.11.4.6, weak_ptr specialized algorithms
template<class T>
  void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// 23.11.5, class template owner_less
template<class T = void> struct owner_less;
// 23.11.6, class template enable_shared_from_this
template<class T> class enable_shared_from_this;

// 23.11.7, 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>>;

// 23.11.8, atomic smart pointers
template<class T> struct atomic<shared_ptr<T>>;
template<class T> struct atomic<weak_ptr<T>>;

// 23.10.8.1, uses_allocator
template<class T, class Alloc>
inline constexpr bool uses_allocator_v = uses_allocator<T, Alloc>::value;

23.10.3 Pointer traits

The class template `pointer_traits` supplies a uniform interface to certain attributes of pointer-like types.

namespace std {
  template<class Ptr> struct pointer_traits {
    using pointer = Ptr;
    using element_type = see below;
    using difference_type = see below;
    template<class U> using rebind = see below;

    static pointer pointer_to(see below r);
  };

  template<class T> struct pointer_traits<T*> {
    using pointer = T*;
    using element_type = T;
    using difference_type = ptrdiff_t;
    template<class U> using rebind = U*;

    static pointer pointer_to(see below r) noexcept;
  };
}

23.10.3.1 Pointer traits member types

using element_type = see below;

Type: `Ptr::element_type` if the qualified-id `Ptr::element_type` is valid and denotes a type (17.9.2); 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 (17.9.2); 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 (17.9.2); 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.

23.10.3.2 Pointer traits member functions

static pointer pointer_traits::pointer_to(see below r);
static pointer pointer_traits<T*>::pointer_to(see below r) noexcept;

Remarks: If `element_type` is `cv void`, the type of `r` is unspecified; otherwise, it is `element_type`.
Returns: The first member function returns a pointer to \( r \) obtained by calling \( \text{Ptr}::\text{pointer_to}(r) \) through which indirection is valid; an instantiation of this function is ill-formed if \( \text{Ptr} \) does not have a matching \( \text{pointer_to} \) static member function. The second member function returns \( \text{addressof}(r) \).

23.10.3.3 Pointer traits optional members

Specializations of \( \text{pointer_traits} \) may define the member declared in this subclause to customize the behavior of the standard library.

\[
\text{static element_type* to_address(pointer p) noexcept;}
\]

Returns: A pointer of type \( \text{element_type*} \) that references the same location as the argument \( p \).

[Note: This function should be the inverse of \( \text{pointer_to} \). If defined, it customizes the behavior of the non-member function \( \text{to_address} \) (23.10.4). —end note]

23.10.4 Pointer conversion

\[
\text{template<class \text{Ptr}> auto to_address(const \text{Ptr}& p) noexcept;}
\]

Returns: \( \text{pointer_traits<\text{Ptr}>::to_address(p)} \) if that expression is well-formed (see 23.10.3.3), otherwise \( \text{to_address(p.operator->())} \).

\[
\text{template<class \text{T> constexpr T* to_address(T* p) noexcept;}
\]

Requires: \( \text{T} \) is not a function type. Otherwise the program is ill-formed.

Returns: \( p \).

23.10.5 Pointer safety

A complete object is \textit{declared reachable} while the number of calls to \( \text{declare_reachable} \) with an argument referencing the object exceeds the number of calls to \( \text{undeclare_reachable} \) with an argument referencing the object.

\[
\text{void declare_reachable(void* p);} \tag{23.10.5.1}
\]

Requires: \( p \) shall be a safely-derived pointer (6.6.4.4.3) or a null pointer value.

Effects: If \( p \) is not null, the complete object referenced by \( p \) is subsequently declared reachable (6.6.4.4.3).

Throws: May throw \text{bad_alloc} if the system cannot allocate additional memory that may be required to track objects declared reachable.

\[
\text{template<class \text{T>} \text{T* undeclare_reachable(T* p);} \tag{23.10.5.5}
\]

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 \( \text{undeclare_reachable(p)} \) call on the object.

Returns: A safely derived copy of \( p \) which shall compare equal to \( p \).

Throws: Nothing.

[Note: It is expected that calls to \( \text{declare_reachable(p)} \) will consume a small amount of memory in addition to that occupied by the referenced object until the matching call to \( \text{undeclare_reachable(p)} \) is encountered. Long running programs should arrange that calls are matched. —end note]

\[
\text{void declare_no_pointers(char* p, size_t n);} \tag{23.10.5.9}
\]

Requires: No bytes in the specified range are currently registered with \( \text{declare_no_pointers()} \). 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 \( \text{undeclare_no_pointers()} \) call. [Note: In a garbage-collecting implementation, the fact that a region in an object is registered with \( \text{declare_no_pointers()} \) 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 \text{operator new} and not previously declared reachable. [Note: This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. —end note]

Throws: Nothing.
void undeclare_no_pointers(char* p, size_t n);

Requires: The same range shall previously have been passed to declare_no_pointers().

Effects: Unregisters a range registered with declare_no_pointers() for destruction. It shall be called before the lifetime of the object ends.

Throws: Nothing.

pointer_safety get_pointer_safety() noexcept;

Returns: pointer_safety::strict if the implementation has strict pointer safety (6.6.4.4.3). It is implementation-defined whether get_pointer_safety returns pointer_safety::relaxed or pointer_safety::preferred if the implementation has relaxed pointer safety.\(^{225}\)

23.10.6 Align

void* align(size_t alignment, size_t size, void*& ptr, size_t& space);

Effects: If it is possible to fit size bytes of storage aligned by alignment into the buffer pointed to by ptr with length space, the function updates ptr to represent the first possible address of such storage and decreases space by the number of bytes used for alignment. Otherwise, the function does nothing.

Requires:

1. alignment shall be a power of two
2. ptr shall represent the address of contiguous storage of at least space bytes

Returns: A null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of ptr.

[Note: The function updates its ptr and space arguments so that it can be called repeatedly with possibly different alignment and size arguments for the same buffer. — end note]

23.10.7 Allocator argument tag

namespace std {
    struct allocator_arg_t { explicit allocator_arg_t() = default; 
    inline constexpr allocator_arg_t allocator_arg{};
    }
}

The allocator_arg_t struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see tuple 23.5) have constructors with allocator_arg_t as the first argument, immediately followed by an argument of a type that satisfies the Allocator requirements (20.5.3.5).

23.10.8 uses_allocator

23.10.8.1 uses_allocator trait

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 BinaryTypeTrait requirements (23.15.1). 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 (17.9.2) 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 user-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.

\(^{225}\) pointer_safety::preferred might be returned to indicate that a leak detector is running so that the program can avoid spurious leak reports.
23.10.8.2 Uses-allocator construction

Uses-allocator construction with allocator `Alloc` refers to the construction of an object `obj` of type `T`, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, ..., VN`, respectively, and an allocator `alloc` of type `Alloc`, according to the following rules:

1. If `uses_allocator_v<T, Alloc>` is false and `is_constructible_v<T, V1, V2, ..., VN>` is true, then `obj` is initialized as `obj(v1, v2, ..., vN);`
2. Otherwise, if `uses_allocator_v<T, Alloc>` is true and `is_constructible_v<T, allocator_arg_t, Alloc, V1, V2, ..., VN>` is true, then `obj` is initialized as `obj(allocator_arg, alloc, v1, v2, ..., vN);`
3. Otherwise, if `uses_allocator_v<T, Alloc>` is true and `is_constructible_v<T, V1, V2, ..., VN, Alloc>` is true, then `obj` is initialized as `obj(v1, v2, ..., vN, alloc);`
4. Otherwise, the request for uses-allocator construction is ill-formed. [Note: An error will result if `uses_allocator_v<T, Alloc>` is true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass the allocator to an element. —end note]

23.10.9 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]

```cpp
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);
    };
}
```

§ 23.10.9
23.10.9.1 Allocator traits member types

using pointer = see below;

Type: Alloc::pointer if the qualified-id Alloc::pointer is valid and denotes a type (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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 (17.9.2); 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.

23.10.9.2 Allocator traits 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)...).

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; otherwise, rhs.

23.10.10  The default allocator

  All specializations of the default allocator satisfy the allocator completeness requirements (20.5.3.5.1).

namespace std {
  template<class T> class allocator {
    public:
      using value_type = T;
      using propagate_on_container_move_assignment = true_type;
      using is_always_equal = true_type;
      allocator() noexcept;
      allocator(const allocator&) noexcept;
      template<class U> allocator(const allocator<U>&) noexcept;
      ~allocator();
      [[nodiscard]] T* allocate(size_t n);
      void deallocate(T* p, size_t n);
  };
}

23.10.10.1  allocator 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.

[[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 (21.6.2), but it is unspecified when or how often this function is called.

  Throws: bad_alloc if the storage cannot be obtained.

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 (21.6.2), but it is unspecified when this function is called.
23.10.10.2 allocator globals

```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.
```

23.10.11 Specialized algorithms

Throughout this subclause, the names of template parameters are used to express type requirements.

1. If an algorithm's template parameter is named `InputIterator`, the template argument shall satisfy the requirements of an input iterator (27.2.3).

2. If an algorithm's template parameter is named `ForwardIterator`, the template argument shall satisfy the requirements of a forward iterator (27.2.5), 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 there are no effects.

23.10.11.1 addressof

```cpp
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 (20.3.6) if E is an lvalue constant subexpression.

23.10.11.2 uninitialized_default_construct

```cpp
template<class ForwardIterator>
void uninitialized_default_construct(ForwardIterator first, ForwardIterator last);

Effects: Equivalent to:
```
23.10.11.4 uninitialized_copy

```cpp
template<class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                  ForwardIterator result);
```

1. **Effects:** As if by:
   ```cpp
   for (; first != last; ++result, (void) ++first)
       ::new (static_cast<void*>(addressof(*result)))
           typename iterator_traits<ForwardIterator>::value_type(*first);
   ```

2. **Returns:** `result`.

**template<class InputIterator, class Size, class ForwardIterator>**
ForwardIterator uninitialized_copy_n(InputIterator first, Size n, ForwardIterator result);

3. **Effects:** As if by:
   ```cpp
   for ( ; n > 0; ++result, (void) ++first, --n) {
       ::new (static_cast<void*>(addressof(*result)))
           typename iterator_traits<ForwardIterator>::value_type(*first);
   }
   ```

4. **Returns:** `result`.

23.10.11.5 uninitialized_move

```cpp
template<class InputIterator, class ForwardIterator>
ForwardIterator uninitialized_move(InputIterator first, InputIterator last,
                                  ForwardIterator result);
```

1. **Effects:** Equivalent to:
   ```cpp
   for (; first != last; (void)+result, ++first)
       ::new (static_cast<void*>(addressof(*result)))
           typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
   return result;
   ```

2. **Remarks:** If an exception is thrown, some objects in the range `[first, last)` are left in a valid but unspecified state.

**template<class InputIterator, class Size, class ForwardIterator>**
pair<InputIterator, ForwardIterator>  
uninitialized_move_n(InputIterator first, Size n, ForwardIterator result);

3. **Effects:** Equivalent to:
   ```cpp
   for (; n > 0; ++result, (void) ++first, --n) {
       ::new (static_cast<void*>(addressof(*result)))
           typename iterator_traits<ForwardIterator>::value_type(std::move(*first));
   return {first,result};
   }
   ```

4. **Remarks:** If an exception is thrown, some objects in the range `[first, std::next(first,n))` are left in a valid but unspecified state.

23.10.11.6 uninitialized_fill

```cpp
template<class ForwardIterator, class T>
void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x);
```

1. **Effects:** As if by:
   ```cpp
   for (; first != last; ++first)
       ::new (static_cast<void*>(addressof(*first)))
           typename iterator_traits<ForwardIterator>::value_type(x);
   ```

**template<class ForwardIterator, class Size, class T>**
ForwardIterator uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

2. **Effects:** As if by:
for (; n--; ++first)
    ::new (static_cast<void*>(addressof(*first)))
        typename iterator_traits<ForwardIterator>::value_type(x);

return first;

23.10.11.7 destroy [specialized.destroy]

template<class T>
void destroy_at(T* location);

1 Effects: Equivalent to:
        location->~T();

template<class ForwardIterator>
void destroy(ForwardIterator first, ForwardIterator last);

2 Effects: Equivalent to:
        for (; first!=last; ++first)
        destroy_at(addressof(*first));

template<class ForwardIterator, class Size>
ForwardIterator destroy_n(ForwardIterator first, Size n);

3 Effects: Equivalent to:
        for (; n > 0; (void)++first, --n)
        destroy_at(addressof(*first));

return first;

23.10.12 C library memory allocation [c.malloc]

1 [ Note: The header <cstdlib> (21.2.2) 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);

2 Effects: These functions have the semantics specified in the C standard library.

3 Remarks: These functions do not attempt to allocate storage by calling ::operator new() (21.6).

4 Storage allocated directly with these functions is implicitly declared reachable (see 6.6.4.4.3) 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);

5 Effects: This function has the semantics specified in the C standard library.

6 Remarks: This function does not attempt to deallocate storage by calling ::operator delete().

See also: ISO C 7.22.3

23.11 Smart pointers [smartptr]

23.11.1 Class template unique_ptr [unique.ptr]

1 A unique pointer is an object that owns another object and manages that other object through a pointer. More precisely, a unique pointer is an object u that stores a pointer to a second object p and will dispose of p when u is itself destroyed (e.g., when leaving block scope (9.7)). In this context, u is said to own p.

2 The mechanism by which u disposes of p is known as p’s associated deleter, a function object whose correct invocation results in p’s appropriate disposition (typically its deletion).
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.

Additionally, \( u \) can, upon request, transfer ownership to another unique pointer \( u2 \). Upon completion of such a transfer, the following postconditions hold:

1. \( u2.p \) is equal to the pre-transfer \( u.p \),
2. \( u.p \) is equal to \( \text{nullptr} \), and
3. if the pre-transfer \( u.d \) maintained state, such state has been transferred to \( u2.d \).

As in the case of a reset, \( u2 \) properly disposes of its pre-transfer owned object via the pre-transfer associated deleter before the ownership transfer is considered complete. [Note: A deleter’s state need never be copied, only moved or swapped as ownership is transferred. — end note]

Each object of a type \( U \) instantiated from the \textit{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 \textit{MoveConstructible} and \textit{MoveAssignable}, but is not \textit{CopyConstructible} nor \textit{CopyAssignable}. The template parameter \( T \) of \textit{unique_ptr} may be an incomplete type.

[Note: The uses of \textit{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]

### 23.11.1.1 Default deleters

#### 23.11.1.1.1 In general

The class template \textit{default_delete} serves as the default deleter (destruction policy) for the class template \textit{unique_ptr}.

The template parameter \( T \) of \textit{default_delete} may be an incomplete type.

### 23.11.1.2 \textit{default_delete}

```cpp
namespace std {
    template<class T> struct default_delete {
        constexpr default_delete() noexcept = default;
        template<class U> default_delete(const default_delete<U>&) noexcept;
        template<class U> void operator()(T* ptr) const;
    };
}
```

#### Effects

1. Constructs a \textit{default_delete} object from another \textit{default_delete\<U\>} object.

#### Remarks

1. This constructor shall not participate in overload resolution unless \( U* \) is implicitly convertible to \( T* \).

2. Calls \textit{delete} on \( ptr \).

3. If \( T \) is an incomplete type, the program is ill-formed.

### 23.11.1.3 \textit{default_delete\<T[]\>}

```cpp
namespace std {
    template<class T> struct default_delete<T[]> {
        constexpr default_delete() noexcept = default;
        template<class U> default_delete(const default_delete<U[]>&) noexcept;
        template<class U> void operator()(U* ptr) const;
    };
}
```

#### Effects

1. Constructs a \textit{default_delete} object from another \textit{default_delete\<U[]\>} object.

#### Remarks

1. This constructor shall not participate in overload resolution unless \( U(*)[] \) is convertible to \( T(*)[] \).
template<class U> 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(*)[].

23.11.1.2 unique_ptr for single objects [unique.ptr.single]

namespace std {
    template<class T, class D = default_delete<T>> class unique_ptr {
        public:
            using pointer = remove_reference_t<T>;
            using element_type = T;
            using deleter_type = D;

            // 23.11.1.2.1, constructors
            constexpr unique_ptr() noexcept;
            explicit unique_ptr(pointer p) noexcept;
            unique_ptr(pointer p, remove_reference_t<D>) noexcept;
            unique_ptr(pointer p, remove_reference_t<D> 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;

            // 23.11.1.2.2, destructor
            ~unique_ptr();

            // 23.11.1.2.3, assignment
            unique_ptr& operator=(unique_ptr&& u) noexcept;
            template<class U, class E>
                unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;
            unique_ptr& operator=(nullptr_t) noexcept;

            // 23.11.1.2.4, observers
            add_lvalue_reference_t<T> operator*() const;
            pointer operator->() const noexcept;
            pointer get() const noexcept;
            deleter_type& get_deleter() noexcept;
            const deleter_type& get_deleter() const noexcept;
            explicit operator bool() const noexcept;

            // 23.11.1.2.5, modifiers
            pointer release() noexcept;
            void reset(pointer p = pointer()) noexcept;
            void swap(unique_ptr& u) noexcept;

            // disable copy from lvalue
            unique_ptr(const unique_ptr&) = delete;
            unique_ptr& operator=(const unique_ptr&) = delete;
    }
}

1 The default type for the template parameter D is default_delete. A client-supplied template argument D shall be a function object type (23.14), lvalue reference to function, or lvalue reference to function object type for which, given a value d of type D and a value ptr of type unique_ptr<T, D>::pointer, the expression d(ptr) is valid and has the effect of disposing of the pointer as appropriate for that deleter.

2 If the deleter’s type D is not a reference type, D shall satisfy the requirements of Destructible (Table 27).

3 If the qualified-id remove_reference_t<T>::pointer is valid and denotes a type (17.9.2), then unique_ptr<T, D>::pointer shall be a synonym for remove_reference_t<T>::pointer. Otherwise unique_ptr<T, D>::pointer shall be a synonym for element_type*. The type unique_ptr<T, D>::pointer shall satisfy the requirements of NullablePointer (20.5.3.3).
[Example: Given an allocator type \( X \) (20.5.3.5) and letting \( A \) be a synonym for \( \text{allocator_traits}\langle X \rangle \), the types \( A::\text{pointer} \), \( A::\text{const_pointer} \), \( A::\text{void_pointer} \), and \( A::\text{const_void_pointer} \) may be used as unique_ptr<T, D>::\text{pointer}. — end example]

23.11.1.2.1 unique_ptr constructors

\begin{verbatim}
constexpr unique_ptr() noexcept;
constexpr unique_ptr(nullptr_t) noexcept;

Requires: \( D \) shall satisfy the requirements of \texttt{DefaultConstructible} (Table 22), and that construction shall not throw an exception.

Effects: Constructs a unique_ptr object that owns nothing, value-initializing the stored pointer and the stored deleter.

Postconditions: \texttt{get()} == nullptr.get_deleter() returns a reference to the stored deleter.

Remarks: If is_pointer_v<deleter_type> is true or is_default_constructible_v<deleter_type> is false, this constructor shall not participate in overload resolution.
\end{verbatim}

\begin{verbatim}
explicit unique_ptr(pointer p) noexcept;
unique_ptr(pointer p, see below d1) noexcept;
unique_ptr(pointer p, see below d2) noexcept;

The signature of these constructors depends upon whether \( D \) is a reference type. If \( D \) is a non-reference type \( A \), then the signatures are:
unique_ptr(pointer p, const A&) noexcept;
unique_ptr(pointer p, A&& d) noexcept;

If \( D \) is an lvalue reference type \( A& \), then the signatures are:
unique_ptr(pointer p, A& d) noexcept;
unique_ptr(pointer p, A&& d) = delete;

If \( D \) is an lvalue reference type const \( A& \), then the signatures are:
unique_ptr(pointer p, const A& d) noexcept;
unique_ptr(pointer p, const A&& d) = delete;

Requires: For the first constructor, if \( D \) is not a reference type, \( D \) shall satisfy the requirements of \texttt{CopyConstructible} and such construction shall not exit via an exception. For the second constructor, if \( D \) is not a reference type, \( D \) shall satisfy the requirements of \texttt{MoveConstructible} and such construction shall not exit via an exception.

Effects: Constructs a unique_ptr object which owns \( p \), initializing the stored pointer with \( p \) and value-initializing the stored deleter from std::forward<decltype(d)>(d).

Remarks: These constructors shall not participate in overload resolution unless is_constructible_v<D, decltype(d)> is true.

Postconditions: \texttt{get()} == p.get_deleter() returns a reference to the stored deleter. If \( D \) is a reference type then get_deleter() returns a reference to the lvalue \( d \).

Remarks: If class template argument deduction (16.3.1.8) would select a function template corresponding to either of these constructors, then the program is ill-formed.

[Example:

\begin{verbatim}
D d;
\end{verbatim}

§ 23.11.1.2.1
unique_ptr(int, D) p1(new int, D()); // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d); // D must be CopyConstructible
unique_ptr<int, D&> p3(new int, d); // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined with reference deleter type

— end example

unique_ptr(unique_ptr&& u) noexcept;

Requires: If D is not a reference type, D shall satisfy the requirements of MoveConstructible (Table 23). Construction of the deleter from an rvalue of type D shall not throw an exception.

Effects: Constructs a unique_ptr by transferring ownership from u to *this. If D is a reference type, this deleter is copy constructed from u’s deleter; otherwise, this deleter is move constructed from u’s deleter. [Note: The deleter constructor can be implemented with std::forward<D>. — end note]

Postconditions: get() yields the value u.get() yielded before the construction. get_deleter() returns a reference to the stored deleter that was constructed from u.get_deleter(). If D is a reference type then get_deleter() and u.get_deleter() both reference the same lvalue deleter.

deleted

template<class U, class E> unique_ptr(unique_ptr<U, E>&& u) noexcept;

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.

Remarks: This constructor shall not participate in overload resolution unless:

(22.1) unique_ptr<U, E>::pointer is implicitly convertible to pointer,
(22.2) U is not an array type, and
(22.3) either D is a reference type and E is the same type as D, or D is not a reference type and E is implicitly convertible to D.

Effects: Constructs a unique_ptr by transferring ownership from u to *this. If E is a reference type, this deleter is copy constructed from u’s deleter; otherwise, this deleter is move constructed from u’s deleter. [Note: The deleter constructor can be implemented with std::forward<E>. — end note]

Postconditions: get() yields the value u.get() yielded before the construction. get_deleter() returns a reference to the stored deleter that was constructed from u.get_deleter().

23.11.1.2.2 unique_ptr destructor

~unique_ptr();

Requires: If E is not a reference type, 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()).

23.11.1.2.3 unique_ptr assignment

unique_ptr& operator=(unique_ptr&& u) noexcept;

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 not throw an exception.

Effects: Transfers ownership from u to *this as if by calling reset(u.release()) followed by get_deleter() = std::forward<D>(u.get_deleter()).

Returns: *this.

deleted

template<class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u) noexcept;

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:

(5.1) unique_ptr<U, E>::pointer is implicitly convertible to pointer, and
(5.2) U is not an array type, and
(5.3) is_assignable_v<D&, E&&> is true.

Effects: Transfers ownership from u to *this as if by calling reset(u.release()) followed by get_deleter() = std::forward<E>(u.get_deleter()).

Returns: *this.

unique_ptr& operator=(nullptr_t) noexcept;

Effects: As if by reset().

Postconditions: get() == nullptr.

Returns: *this.

23.11.1.2.4 unique_ptr observers

add_lvalue_reference_t<T> operator*() const;

Requires: get() != nullptr.

Returns: *get().

pointer operator->() const noexcept;

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;

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.

23.11.1.2.5 unique_ptr modifiers

pointer release() noexcept;

Postconditions: 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]

Postconditions: get() == p. [Note: The postcondition does not hold if the call to get_deleter() destroys *this since this->get() is no longer a valid expression. —end note]

void swap(unique_ptr& u) noexcept;

Requires: get_deleter() shall be swappable (20.5.3.2) and shall not throw an exception under swap.

Effects: Invokes swap on the stored pointers and on the stored deleters of *this and u.
23.11.1.3 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;

            // 23.11.1.3.1, 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;
            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;

            // 23.11.1.3.3, 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;

            // 23.11.1.3.4, modifiers
            pointer release() noexcept;
            template<class U> void reset(U p) noexcept;
            void reset(nullptr_t = nullptr) noexcept;
            void swap(unique_ptr& u) noexcept;

            // disable copy from lvalue
            unique_ptr(const unique_ptr&) = delete;
            unique_ptr& operator=(const unique_ptr&) = delete;
    }
}

1 A specialization for array types is provided with a slightly altered interface.

(1.1) — Conversions between different types of unique_ptr<T[], D> that would be disallowed for the corresponding pointer-to-array types, and conversions to or from the non-array forms of unique_ptr, produce an ill-formed program.

(1.2) — Pointers to types derived from T are rejected by the constructors, and by reset.

(1.3) — The observers operator* and operator-> are not provided.

(1.4) — The indexing observer operator[] is provided.

(1.5) — The default deleter will call delete[].

2 Descriptions are provided below only for members that differ from the primary template.

3 The template argument T shall be a complete type.
23.11.1.3.1 unique_ptr 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> unique_ptr(U p, see below d) noexcept;
template<class U> unique_ptr(U p, see below d) noexcept;
```

These constructors behave the same as the constructors in the primary template that take a parameter of type `pointer` and a second parameter except that they shall not participate in overload resolution unless either:

1. `U` is the same type as `pointer`,
2. `U` is `nullptr_t`, or
3. `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]

23.11.1.3.2 unique_ptr assignment

```cpp
template<class U, class E> unique_ptr 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]

23.11.1.3.3 unique_ptr observers

```cpp
T& operator[](size_t i) const;
```

1. `Requires: i < the number of elements in the array to which the stored pointer points.`
2. `Returns: get()[i].`

23.11.1.3.4 unique_ptr modifiers

```cpp
void reset(nullptr_t p = nullptr) noexcept;
```

1. `Effects: Equivalent to reset(pointer()).`
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. \( \text{U} \) is the same type as \( \text{pointer} \), or
2. \( \text{pointer} \) is the same type as \( \text{element_type}* \), \( \text{U} \) is a pointer type \( \text{V}* \), and \( \text{V}(*[] \) is convertible to \( \text{element_type}(*)[] \).

### 23.11.1.4 unique_ptr 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: \( \text{unique_ptr<T>(new T(std::forward<Args>(args)...))} \).

```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: \( \text{unique_ptr<T>(new remove_extent_t<T>[n]())} \).

```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.

### 23.11.1.5 unique_ptr specialized algorithms

```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: \( x\text{.get()} == y\text{.get()} \).

```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: \( x\text{.get()} != y\text{.get()} \).

```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

\[
\text{common_type_t<typename unique_ptr<T1, D1>::pointer,}
\text{typename unique_ptr<T2, D2>::pointer>}
\]

Then the specialization \( \text{less<CT>} \) shall be a function object type (23.14) that induces a strict weak ordering (28.7) on the pointer values.

Returns: \( \text{less<CT>}(x\text{.get()}, y\text{.get()}) \).

Remarks: If \( \text{unique_ptr<T1, D1>::pointer} \) is not implicitly convertible to \( CT \) or \( \text{unique_ptr<T2, D2>::pointer} \) 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: \( \neg (y < x) \).

```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: \( \neg (x < y) \).

§ 23.11.5
template<class T, class D>
bool operator==(const unique_ptr<T, D>& x, nullptr_t) noexcept;

Returns: !x.

template<class T, class D>
bool operator!=(const unique_ptr<T, D>& x, nullptr_t) noexcept;

template<class T, class D>
bool operator%(const unique_ptr<T, D>& x, nullptr_t) noexcept;

template<class T, class D>
bool operator%(nullptr_t, const unique_ptr<T, D>& x) noexcept;

template<class T, class D>
bool operator<(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator<(nullptr_t, const unique_ptr<T, D>& x);

Returns: (bool)x.

template<class T, class D>
bool operator<=(const unique_ptr<T, D>& x, nullptr_t);

template<class T, class D>
bool operator<=(nullptr_t, const unique_ptr<T, D>& x);

Returns: !x.

Returns: The specialization less<unique_ptr<T, D>::pointer> shall be a function object type (23.14) that induces a strict weak ordering (28.7) on the pointer values.

Returns: The first function template returns less<unique_ptr<T, D>::pointer>()(x.get(), nullptr)

The second function template returns
less<unique_ptr<T, D>::pointer>()(nullptr, x.get())

Returns: The first function template returns nullptr < x. The second function template returns x < nullptr.

Returns: The first function template returns !(nullptr < x). The second function template returns !(x < nullptr).

Returns: The first function template returns !(x < nullptr). The second function template returns !(nullptr < x).

23.11.1.6 unique_ptr 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.

23.11.2 Class bad_weak_ptr

namespace std {
    class bad_weak_ptr : public exception {
    public:
        bad_weak_ptr() noexcept;
    }
An exception of type \texttt{bad\_weak\_ptr} is thrown by the \texttt{shared\_ptr} constructor taking a \texttt{weak\_ptr}.

\begin{verbatim}
bad\_weak\_ptr() noexcept;
\end{verbatim}

\textbf{Postconditions:} \texttt{what()} returns an implementation-defined NTBS.

### 23.11.3 Class template \texttt{shared\_ptr} \[util.smartptr.shared\]

The \texttt{shared\_ptr} class template stores a pointer, usually obtained via \texttt{new}. \texttt{shared\_ptr} implements semantics of shared ownership: the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A \texttt{shared\_ptr} is said to be empty if it does not own a pointer.

```cpp
namespace std {
  template<class T> class shared_ptr {
    public:
      using element_type = remove_extent_t<T>;
      using weak_type = weak_ptr<T>;
      // 23.11.3.1, constructors
      constexpr shared_ptr() noexcept;
      constexpr shared_ptr(nullptr_t) noexcept : shared_ptr() { }
      template<class Y>
        explicit shared_ptr(Y* p) noexcept : shared_ptr() { }
      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;
      shared_ptr(const shared_ptr& r) noexcept;
      template<class Y>
        shared_ptr(const shared_ptr<Y>& r) noexcept;
      shared_ptr(shared_ptr&& r) noexcept;
      template<class Y>
        shared_ptr(shared_ptr<Y>&& r) noexcept;
      template<class Y, class D>
        shared_ptr(unique_ptr<Y, D>&& r);
      // 23.11.3.2, destructor
      ~shared_ptr();
      // 23.11.3.3, 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);
      // 23.11.3.4, modifiers
      void swap(shared_ptr& r) noexcept;
      void reset() noexcept;
      template<class Y>
        void reset(Y* p);
  }
}
```

§ 23.11.3
template<class Y, class D>
    void reset(Y* p, D d);
#endif

template<class Y, class D, class A>
    void reset(Y* p, D d, A a);
#endif

// 23.11.3.5, 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>;

template<class T, class D>
    shared_ptr(unique_ptr<T, D>) -> shared_ptr<T>;

2 Specializations of shared_ptr shall be CopyConstructible, CopyAssignable, and LessThanComparable, allowing their use in standard containers. Specializations of shared_ptr shall be contextually convertible to bool, allowing their use in boolean expressions and declarations in conditions. The template parameter T of shared_ptr may be an incomplete type.

3 [Example:

        if (shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
            // do something with px
        }
    —end example]

4 For purposes of determining the presence of a data race, member functions shall access and modify only the shared_ptr and weak_ptr objects themselves and not objects they refer to. Changes in use_count() do not reflect modifications that can introduce data races.

5 For the purposes of subclause 23.11, a pointer type Y* is said to be compatible with a pointer type T* when either Y* is convertible to T* or Y is U[N] and T is cv U[].

23.11.3.1 shared_ptr constructors [util.smartptr.shared.const]

In the constructor definitions below, enables shared_from_this with p, for a pointer p of type Y*, means that if Y has an unambiguous and accessible base class that is a specialization of enable_shared_from_this (23.11.6), then remove_cv_t<T*> shall be implicitly convertible to T* and the constructor evaluates the statement:

        if (p != nullptr && p->weak_this.expired())
            p->weak_this = shared_ptr<remove_cv_t<T*>>(*this, const_cast<remove_cv_t<T*>*>(p));

The assignment to the weak_this member is not atomic and conflicts with any potentially concurrent access to the same object (6.8.2).

constexpr shared_ptr() noexcept;

    Effects: Constructs an empty shared_ptr object.

    Postconditions: use_count() == 0 && get() == nullptr.

template<class Y> explicit shared_ptr(Y* p);

    Requires: Y shall be a complete type. The expression delete[] p, when T is an array type, or delete p, when T is not an array type, shall have well-defined behavior, and shall not throw exceptions.

    Effects: When T is not an array type, constructs a shared_ptr object that owns the pointer p. Otherwise, constructs a shared_ptr that owns p and a deleter of an unspecified type that calls delete[] p. When
T is not an array type, enables shared_from_this with p. If an exception is thrown, delete p is called when T is not an array type, delete[] p otherwise.

Postconditions: use_count() == 1 && get() == p.

Throws: bad_alloc, or an implementation-defined exception when a resource other than memory could not be obtained.

Remarks: When T is an array type, this constructor shall not participate in overload resolution unless the expression delete[] p is well-formed and either T is U[N] and Y(*)[N] is convertible to T*, or T is U[] and Y(*)[] is convertible to T*. When T is not an array type, this constructor shall not participate in overload resolution unless the expression delete p is well-formed and Y* is convertible to T*.

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);

Requires: Construction of d and a deleter of type D initialized with std::move(d) shall not throw exceptions. The expression d(p) shall have well-defined behavior and shall not throw exceptions. A shall be an allocator (20.5.3.5).

Effects: Constructs a shared_ptr object that owns the object p and the deleter d. When T is not an array type, the first and second constructors enable shared_from_this with p. The second and fourth constructors shall use a copy of a to allocate memory for internal use. If an exception is thrown, d(p) is called.

Postconditions: use_count() == 1 && get() == p.

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*.

template<class Y> shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;

Effects: Constructs a shared_ptr instance that stores p and shares ownership with r.

Postconditions: get() == p && use_count() == r.use_count().

[Note: To avoid the possibility of a dangling pointer, the user of this constructor should ensure that p remains valid at least until the ownership group of r is destroyed. —end note]

[Note: This constructor allows creation of an empty shared_ptr instance with a non-null stored pointer. —end note]

shared_ptr(const shared_ptr& r) noexcept;

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.

Postconditions: get() == r.get() && use_count() == r.use_count().

shared_ptr(shared_ptr&& 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.

Postconditions: *this shall contain the old value of r. r shall be empty. r.get() == nullptr.
template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);

Effects: Constructs a shared_ptr object that shares ownership with \( r \) and stores a copy of the pointer stored in \( r \). If an exception is thrown, the constructor has no effect.

Postconditions: use_count() == r.use_count().

Throws: bad_weak_ptr when \( r.expired() \).

Remarks: This constructor shall not participate in overload resolution unless \( Y* \) is compatible with \( T* \).

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.

23.11.3.2 shared_ptr destructor

~shared_ptr();

Effects:

1. If \( \ast\this\) is empty or shares ownership with another \( \ast\shared\_ptr\) instance (use_count() > 1), there are no side effects.
2. Otherwise, if \( \ast\this\) owns an object \( p \) and a deleter \( d\), \( d(p) \) is called.
3. Otherwise, \( \ast\this\) owns a pointer \( p \), and delete \( p \) is called.

[Note: Since the destruction of \( \ast\this\) decreases the number of instances that share ownership with \( \ast\this\) by one, after \( \ast\this\) has been destroyed all \( \shared\_ptr\) instances that shared ownership with \( \ast\this\) will report a use_count() that is one less than its previous value. — end note]

23.11.3.3 shared_ptr 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: \( \ast\this\).

[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:

\begin{verbatim}
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
\end{verbatim}

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: \( \ast\this\).

\( \shared\_ptr\) operator=(unique_ptr<Y, D>&& \( r \));

Effects: Equivalent to \( \shared\_ptr(\std::move(r)).\swap(*\this) \).

Returns: \( \ast\this\).

23.11.3.4 shared_ptr modifiers

void swap(shared_ptr& \( r \)) noexcept;

Effects: Exchanges the contents of \( \ast\this\) 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).

23.11.3.5 shared_ptr observers

element_type* get() const noexcept;

Returns: The stored pointer.

T& operator*() const noexcept;

Requires: get() != 0.

Returns: *get().

Remarks: When T is an array type or cv void, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

T* operator->() const noexcept;

Requires: get() != 0.

Returns: get().

Remarks: When T is an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

element_type& operator[](ptrdiff_t i) const;

Requires: get() != 0 && i >= 0. If T is U[N], i < N.

Returns: get()[i].

Remarks: When T is not an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well-formed.

Throws: Nothing.

long use_count() const noexcept;

Returns: The number of shared_ptr objects, *this included, that share ownership with *this, or 0 when *this is empty.

Synchronization: None.

[Note: get() == nullptr does not imply a specific return value of use_count(). — end note]

[Note: weak_ptr<T>::lock() can affect the return value of use_count(). — end note]

[Note: When multiple threads can affect the return value of use_count(), the result should be treated as approximate. In particular, use_count() == 1 does not imply that accesses through a previously destroyed shared_ptr have in any sense completed. — end note]

explicit operator bool() const noexcept;

Returns: get() != 0.

template<class U> bool owner_before(const shared_ptr<U>& b) const noexcept;

template<class U> bool owner_before(const weak_ptr<U>& b) const noexcept;

Returns: An unspecified value such that
x.owner_before(y) defines a strict weak ordering as defined in 28.7;
under the equivalence relation defined by owner_before, !a.owner_before(b) &amp; !b.owner_before(a), two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.

23.11.3.6 shared_ptr creation

The common requirements that apply to all make_shared and allocate_shared 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&amp; a, args);

Requires: A shall be an allocator (20.5.3.5).

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 templates use a copy of a (rebound for an unspecified value_type) to allocate memory. If an exception is thrown, the functions have no effect.

Returns: A shared_ptr instance that stores and owns the address of the newly constructed object.

Postconditions: r.get() != 0 &amp; r.use_count() == 1, where r is the return value.

Throws: bad_alloc, or an exception thrown from allocate or from the initialization of the object.

Remarks:

(7.1) — Implementations should perform no more than one memory allocation. [Note: This provides efficiency equivalent to an intrusive smart pointer. — end note]

(7.2) — When an object of an array type U is specified to have an initial value of u (of the same type), this shall be interpreted to mean that each array element of the object has as its initial value the corresponding element from u.

(7.3) — When an object of an array type is specified to have a default initial value, this shall be interpreted to mean that each array element of the object has a default initial value.

(7.4) — When a (sub)object of a non-array type U is specified to have an initial value of v, or U(1...), where 1... is a list of constructor arguments, make_shared shall initialize this (sub)object via the expression ::new(pv) U(v) or ::new(pv) U(1...) respectively, where pv has type void* and points to storage suitable to hold an object of type U.

(7.5) — When a (sub)object of a non-array type U is specified to have an initial value of v, or U(1...), where 1... is a list of constructor arguments, allocate_shared shall initialize this (sub)object via the expression

(7.5.1) — allocator_traits&lt;A2>::construct(a2, pv, v) or

(7.5.2) — allocator_traits&lt;A2>::construct(a2, pv, 1...)

respectively, where pv points to storage suitable to hold an object of type U and a2 of type A2 is a rebound copy of the allocator a passed to allocate_shared such that its value_type is 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&lt;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 U.

(7.8) — Array elements are initialized in ascending order of their addresses.

(7.9) — 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 should be destroyed in the reverse order of their construction.
**Note:** These functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as reference counts. — end note

```cpp
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]

```cpp
template<class T> shared_ptr<T> make_shared(size_t N);

Returns: A `shared_ptr` to an object of type `T[N]` with a default initial value, where `T` is `U[0]`.

Remarks: These overloads shall only participate in overload resolution when `T` is of the form `U[0]`.

[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]

```cpp
template<class T, class A>
shared_ptr<T> allocate_shared(const A& a, size_t N);

Returns: A `shared_ptr` to an object of type `T[N]`, where `T` is `U[0]` 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[0]`.

[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]

```cpp
template<class T, class... Args>
shared_ptr<T> make_shared(const remove_extent_t<T>& u);

Returns: A `shared_ptr` to an object of type `T[N]`, where `T` is `U[0]` 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[0]`.

[Example:
```
shared_ptr<double[]> p = make_shared<double[]>(1024, 1.0);
```
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]

23.11.3.7 shared_ptr comparison [util.smartptr.shared.cmp]

template<class T, class U>
    bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
1 Returns: a.get() == b.get().

template<class T, class U>
    bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
2 Returns: less<>()(a.get(), b.get()).

3 [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;
4 Returns: !a.

template<class T>
    bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
5 Returns: (bool)a.

template<class T>
    bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
6 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).

23.11.3.8 shared_ptr specialized algorithms

template<class T>
void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

Effects: Equivalent to a.swap(b).

23.11.3.9 shared_ptr casts

template<class T, class U>
shared_ptr<T> static_pointer_cast(const 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()))

[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>
shared_ptr<T> dynamic_pointer_cast(const 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).
(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>
shared_ptr<T> const_pointer_cast(const 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()))

[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>
shared_ptr<T> reinterpret_pointer_cast(const 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()))

[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]

23.11.3.10 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]

23.11.3.11 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);

Effects: As if by: os << p.get();

Returns: os.

23.11.4 Class template weak_ptr

The weak_ptr class template stores a weak reference to an object that is already managed by a shared_ptr. To access the object, a weak_ptr can be converted to a shared_ptr using the member function lock.

namespace std {
    template<class T> class weak_ptr {
        public:
            using element_type = remove_extent_t<T>;

            // 23.11.4.1, 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;

            // 23.11.4.2, destructor
            ~weak_ptr();

            // 23.11.4.3, assignment
            weak_ptr& operator=(const weak_ptr& r) noexcept;
            template<class Y>
                weak_ptr& operator=(const weak_ptr<Y>& r) noexcept;
            template<class Y>
                weak_ptr& operator=(const shared_ptr<Y>& r) noexcept;
            weak_ptr& operator=(weak_ptr& r) noexcept;
            template<class Y>
                weak_ptr& operator=(weak_ptr<Y>& r) noexcept;

            // 23.11.4.4, modifiers
            void swap(weak_ptr& r) noexcept;
            void reset() noexcept;
        } }
Specializations of `weak_ptr` shall be `CopyConstructible` and `CopyAssignable`, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

### 23.11.4.1 `weak_ptr` constructors

#### `constexpr weak_ptr()` noexcept;

- **Effects:** Constructs an empty `weak_ptr` object.
- **Postconditions:** `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`.
- **Postconditions:** `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`.
- **Postconditions:** `*this` shall contain the old value of `r`. `r` shall be empty. `r.use_count()` == 0.

### 23.11.4.2 `weak_ptr` destructor

#### `~weak_ptr();`

- **Effects:** Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.

### 23.11.4.3 `weak_ptr` assignment

#### `weak_ptr& operator=(const weak_ptr& r)` noexcept;

- **Effects:** Equivalent to `weak_ptr(r).swap(*this)`.
- **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;
template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;

**Effects:** Equivalent to weak_ptr(std::move(r)).swap(*this).

**Returns:** *this.

### 23.11.4.4 weak_ptr modifiers

void swap(weak_ptr& r) noexcept;

**Effects:** Exchanges the contents of *this and r.

void reset() noexcept;

**Effects:** Equivalent to weak_ptr().swap(*this).

### 23.11.4.5 weak_ptr observers

long use_count() const noexcept;

**Returns:** 0 if *this is empty; otherwise, the number of shared_ptr instances that share ownership with *this.

bool expired() const noexcept;

**Returns:** use_count() == 0.

shared_ptr<T> lock() const noexcept;

**Returns:** expired() ? shared_ptr<T>() : shared_ptr<T>(*this), executed atomically.

**template<class U> bool owner_before(const shared_ptr<U>& b) const noexcept;**

**template<class U> bool owner_before(const weak_ptr<U>& b) const noexcept;**

**Returns:** An unspecified value such that

(4.1) \( x.\text{owner\_before}(y) \) defines a strict weak ordering as defined in 28.7;

(4.2) under the equivalence relation defined by owner_before, !a.owner_before(b) && !b.owner_before(a), two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both empty.

### 23.11.4.6 weak_ptr specialized algorithms

**template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;**

**Effects:** Equivalent to a.swap(b).

### 23.11.5 Class template owner_less

The class template owner_less allows ownership-based mixed comparisons of shared and weak pointers.

**namespace std {**

**template<class T = void> struct owner_less;**

**template<class T> struct owner_less<shared_ptr<T>> {**

**bool operator()(const shared_ptr<T>& a, const shared_ptr<T>& b) const noexcept;**

**bool operator()(const shared_ptr<T>& a, const weak_ptr<T>& b) const noexcept;**

**bool operator()(const weak_ptr<T>& a, const shared_ptr<T>& b) const noexcept;**

**};**

**template<class T> struct owner_less<weak_ptr<T>> {**

**bool operator()(const weak_ptr<T>& a, const weak_ptr<T>& b) const noexcept;**

**bool operator()(const shared_ptr<T>& a, const weak_ptr<T>& b) const noexcept;**

**bool operator()(const weak_ptr<T>& a, const shared_ptr<T>& b) const noexcept;**

**};**

**template<class T, class U>**

**bool operator()(const shared_ptr<T>& a, const shared_ptr<U>& b) 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()(x, y) shall return x.owner_before(y). [Note: Note that

(2.1) operator() defines a strict weak ordering as defined in 28.7;

(2.2) under the equivalence relation defined by operator(), operator()(a, b) && !operator()(b, a),
    two shared_ptr or weak_ptr instances are equivalent if and only if they share ownership or are both
    empty.

—end note]

23.11.6 Class template enable_shared_from_this

A class T can inherit from enable_shared_from_this<T> to inherit the shared_from_this member functions
that obtain a shared_ptr instance pointing to this.

[Example:

struct X: public enable_shared_from_this<X> { }

int main() {
  shared_ptr<X> p(new X);
  shared_ptr<X> q = p->shared_from_this();
  assert(p == q);
  assert(!p.owner_before(q) && !q.owner_before(p)); // p and q share ownership
}

—end example]

namespace std {
  template<class T> class enable_shared_from_this {
    protected:
      constexpr enable_shared_from_this() noexcept;
      enable_shared_from_this(const enable_shared_from_this&) noexcept;
      enable_shared_from_this& operator=(const enable_shared_from_this&) 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
  };
}

3 The template parameter T of enable_shared_from_this may be an incomplete type.

constexpr enable_shared_from_this() noexcept;

4 Effects: Value-initializes weak_this.

5 enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&) noexcept;

6 Returns: *this.

[Note: weak_this is not changed. —end note]
shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;

Returns: shared_ptr<T>(weak_this).

weak_ptr<T> weak_from_this() noexcept;
weak_ptr<T const> weak_from_this() const noexcept;

Returns: weak_this.

23.11.7 Smart pointer hash support [util.smartptr.hash]

template<class T, class D> struct hash<unique_ptr<T, D>>;

Letting UP be unique_ptr<T,D>, the specialization hash<UP> is enabled (23.14.15) if and only if
hash<typename UP::pointer> is enabled. When enabled, for an object \( p \) of type UP, 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()).

23.11.8 Atomic specializations for smart pointers [util.smartptr.atomic]

The library provides partial specializations of the atomic template for shared-ownership smart pointers. The behavior of all operations is as specified in 32.6, 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]

23.11.8.1 Atomic specialization for shared_ptr [util.smartptr.atomic.shared]

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;
}

§ 23.11.8.1 575
private:
    shared_ptr<T> p; // exposition only
};

constexpr atomic() noexcept = default;
1
Effects: Initializes p{}.

atomic(shared_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(shared_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=(shared_ptr<T> desired) noexcept;
5
Effects: Equivalent to store(desired).

shared_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.
7
Effects: Memory is affected according to the value of order.
8
Returns: Atomically returns p.

operator shared_ptr<T>() const noexcept;
9
Effects: Equivalent to: return load();

shared_ptr<T> exchange(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
10
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).
11
Returns: Atomically returns the value of p immediately before the effects.

bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order success, memory_order failure) noexcept;

bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired, memory_order success, memory_order failure) noexcept;

Requires: failure shall not be memory_order::release nor memory_order::acq_rel.
12
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.
13
Returns: true if p was equivalent to expected, false otherwise.
14
Remarks: Two shared_ptr objects are equivalent if they store the same pointer value and either share ownership, or both are empty. The weak form may fail spuriously. See 32.6.1.
15
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(shared_ptr<T>& expected, shared_ptr<T> desired,
    memory_order order = memory_order::seq_cst) noexcept;

Effects: Equivalent to:

    return compare_exchange_weak(expected, desired, order, fail_order);

where fail_order is the same as order except that a value of memory_order::acq_rel shall be replaced by
the value memory_order::acquire and a value of memory_order::release shall be replaced by
the value memory_order::relaxed.

bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
    memory_order order = memory_order::seq_cst) noexcept;

Effects: Equivalent to:

    return compare_exchange_strong(expected, desired, order, fail_order);

where fail_order is the same as order except that a value of memory_order::acq_rel shall be replaced by
the value memory_order::acquire and a value of memory_order::release shall be replaced by
the value memory_order::relaxed.

23.11.8.2 Atomic specialization for weak_ptr

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;

        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
    };}

constexpr atomic() noexcept = default;

Effects: Initializes p{}.

atomic(weak_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]
void store(weak_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.

void operator=(weak_ptr<T> desired) noexcept;

Effects: Equivalent to store(desired).

weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;

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 both are empty. The weak form may fail spuriously. See 32.6.1.

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.

23.12 Memory resources

23.12.1 Header <memory_resource> synopsis

namespace std::pmr {
  // 23.12.2, 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;

  // 23.12.3, class template polymorphic_allocator
  template<class T1> 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;

  // 23.12.4, 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;

  // 23.12.5, pool resource classes
  struct pool_options;
  class synchronized_pool_resource;
  class unsynchronized_pool_resource;
  class monotonic_buffer_resource;
}

23.12.2 Class memory_resource

The 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:
      virtual ~memory_resource();

      [[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;
  };
}
23.12.2.1 memory_resource 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);

23.12.2.2 memory_resource private virtual member functions

virtual void* do_allocate(size_t bytes, size_t alignment) = 0;

Requires: alignment shall be a power of two.

Returns: A derived class shall implement this function to return a pointer to allocated storage (6.6.4.4.1) with a size of at least bytes. The returned storage is aligned to the specified alignment, if such alignment is supported (6.6.5); otherwise it is aligned to max_align.

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;

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.

Effects: A derived class shall implement this function to dispose of allocated storage.

Throws: Nothing.

virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;

Returns: A derived class shall implement this function to return true if memory allocated from this can be deallocated from other and vice-versa, otherwise false. [Note: The most-derived type of other might not match the type of this. For a derived class D, an implementation of this function could immediately return false if dynamic_cast<const D*>(&other) == nullptr. -- end note]

23.12.2.3 memory_resource equality

bool operator==(const memory_resource& a, const memory_resource& b) noexcept;

Returns: &a == &b || a.is_equal(b).

bool operator!=(const memory_resource& a, const memory_resource& b) noexcept;

Returns: !(a == b).

23.12.3 Class template polymorphic_allocator

A specialization of class template pmr::polymorphic_allocator conforms to the Allocator requirements (20.5.3.5). 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.

namespace std::pmr {

  template<class Tp>
  class polymorphic_allocator {
    memory_resource* memory_rsrc; // exposition only

    public:
      using value_type = Tp;
  };

 § 23.12.3 580
// 23.12.3.1, 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;

// 23.12.3.2, member functions
[[nodiscard]] Tp* allocate(size_t n);
deallocate(Tp* p, size_t n);

template<class T, class... Args>
void construct(T* p, Args&&... args);

template<class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct_t,
tuple<Args1...> x, tuple<Args2...> y);

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& pr);

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& pr);

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, U&& x, V&& y);

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& pr);

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& pr);

template<class T>
void destroy(T* p);

polymorphic_allocator select_on_container_copy_construction() const;

memory_resource* resource() const;
};

23.12.3.1 polymorphic_allocator 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().

23.12.3.2 polymorphic_allocator 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.

template<class T, class... Args>
void construct(T* p, Args&&... args);

Requires: Uses_allocator construction of T with allocator resource() (see 23.10.8.2) and constructor arguments std::forward<Args>(args)... is well-formed. [ Note: Uses_allocator construction is always well-formed for types that do not use allocators. — end note ]

Effects: Construct a T object in the storage whose address is represented by p by uses_allocator construction with allocator resource() and constructor arguments std::forward<Args>(args)....

Throws: Nothing unless the constructor for T throws.

template<class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct_t, tuple<Args1...> x, tuple<Args2...> y);

[ Note: This member function and the construct member functions that follow are overloads for piecewise construction of pairs (23.4.2). — end note ]

Effects: Let xprime be a tuple constructed from x according to the appropriate rule from the following list. [ Note: The following description can be summarized as constructing a pair<T1, T2> object in the storage whose address is represented by p, as if by separate uses_allocator construction with allocator resource() (23.10.8.2) of p->first using the elements of x and p->second using the elements of y. — end note ]

(9.1) — If uses_allocator_v<T1,memory_resource*> is false and is_constructible_v<T1,Args1...> is true, then xprime is x.

(9.2) — Otherwise, if uses_allocator_v<T1,memory_resource*> is true and is_constructible_v<T1,allocator_arg_t,memory_resource*,Args1...> is true, then xprime is tuple_cat(make_tuple(allocator_arg, resource())), std::move(x)).

(9.3) — Otherwise, if uses_allocator_v<T1,memory_resource*> is true and is_constructible_v<T1,Args1...,memory_resource*> is true, then xprime is tuple_cat(std::move(x), make_tuple(resource()))).

(9.4) — Otherwise the program is ill formed.

Let yprime be a tuple constructed from y according to the appropriate rule from the following list:

(9.5) — If uses_allocator_v<T2,memory_resource*> is false and is_constructible_v<T2,Args2...> is true, then yprime is y.

(9.6) — Otherwise, if uses_allocator_v<T2,memory_resource*> is true and is_constructible_v<T2,allocator_arg_t,memory_resource*,Args2...> is true, then yprime is tuple_cat(make_tuple(allocator_arg, resource())), std::move(y)).

(9.7) — Otherwise, if uses_allocator_v<T2,memory_resource*> is true and is_constructible_v<T2,Args2...,memory_resource*> is true, then yprime is tuple_cat(std::move(y), make_tuple(resource()))).

(9.8) — Otherwise the program is ill formed.

Then, using piecewise_construct, xprime, and yprime as the constructor arguments, this function constructs a pair<T1, T2> object in the storage whose address is represented by p.

template<class T1, class T2>
void construct(pair<T1, T2>* p);

Effects: Equivalent to:

construct(p, piecewise_construct, tuple<>(), tuple<>());
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, U&& x, V&& y);

Effects: Equivalent to:
           construct(p, piecewise_construct,
                     forward_as_tuple(std::forward<U>(x)),
                     forward_as_tuple(std::forward<V>(y)));

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& pr);

Effects: Equivalent to:
           construct(p, piecewise_construct,
                     forward_as_tuple(pr.first),
                     forward_as_tuple(pr.second));

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& pr);

Effects: Equivalent to:
           construct(p, piecewise_construct,
                     forward_as_tuple(std::forward<U>(pr.first)),
                     forward_as_tuple(std::forward<V>(pr.second)));

template<class T>
void destroy(T* p);

Effects: As if by p->~T().

polymorphic_allocator select_on_container_copy_construction() const;

Returns: polymorphic_allocator().

[ Note: The memory resource is not propagated. — end note ]

memory_resource* resource() const;

Returns: memory_resource.

23.12.3.3 polymorphic_allocator equality

template<class T1, class T2>
bool operator==(const polymorphic_allocator<T1>& a,
               const polymorphic_allocator<T2>& b) noexcept;

Returns: *a.resource() == *b.resource().

template<class T1, class T2>
bool operator!=(const polymorphic_allocator<T1>& a,
               const polymorphic_allocator<T2>& b) noexcept;

Returns: !(a == b).

23.12.4 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.

§ 23.12.4 583
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()`.

```cpp
memory_resource* set_default_resource(memory_resource* r) noexcept;
```

**Effects:** If `r` is non-null, sets the value of the default memory resource pointer to `r`, otherwise sets the default memory resource pointer to `new_delete_resource()`.

**Returns:** The previous value of the default memory resource pointer.

**Remarks:** Calling the `set_default_resource` and `get_default_resource` functions shall not incur a data race. A call to the `set_default_resource` function shall synchronize with subsequent calls to the `set_default_resource` and `get_default_resource` functions.

```cpp
memory_resource* get_default_resource() noexcept;
```

**Returns:** The current value of the default memory resource pointer.

### 23.12.5 Pool resource classes

#### 23.12.5.1 Classes synchronized_pool_resource and unsynchronized_pool_resource

The `synchronized_pool_resource` and `unsynchronized_pool_resource` classes (collectively called pool resource classes) are general-purpose memory resources having the following qualities:

1. **Each resource frees its allocated memory on destruction, even if deallocate has not been called for some of the allocated blocks.**

2. **A pool resource consists of a collection of pools, serving requests for different block sizes. Each individual pool manages a collection of chunks that are in turn divided into blocks of uniform size, returned via calls to do_allocate. Each call to do_allocate(size, alignment) is dispatched to the pool serving the smallest blocks accommodating at least size bytes.**

3. **When a particular pool is exhausted, allocating a block from that pool results in the allocation of an additional chunk of memory from the upstream allocator (supplied at construction), thus replenishing the pool. With each successive replenishment, the chunk size obtained increases geometrically. [Note: By allocating memory in chunks, the pooling strategy increases the chance that consecutive allocations will be close together in memory. —end note]**

4. **Allocation requests that exceed the largest block size of any pool are fulfilled directly from the upstream allocator.**

5. **A pool_options struct may be passed to the pool resource constructors to tune the largest block size and the maximum chunk size.**

A `synchronized_pool_resource` may be accessed from multiple threads without external synchronization and may have thread-specific pools to reduce synchronization costs. An `unsynchronized_pool_resource` class may not be accessed from multiple threads simultaneously and thus avoids the cost of synchronization entirely in single-threaded applications.

```cpp
namespace std::pmr {
    struct pool_options {
        size_t max_blocks_per_chunk = 0;
        size_t largest_required_pool_block = 0;
    };

    class synchronized_pool_resource : public memory_resource {
        public:
            synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

            synchronized_pool_resource() :
                synchronized_pool_resource(pool_options(), get_default_resource()) {};
            explicit synchronized_pool_resource(memory_resource* upstream) :
                synchronized_pool_resource(pool_options(), upstream) {};
            explicit synchronized_pool_resource(const pool_options& opts) :
                synchronized_pool_resource(opts, get_default_resource()) {};
    };
}
```
synchronized_pool_resource(const synchronized_pool_resource&) = delete;
virtual ~synchronized_pool_resource();

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;
};

23.12.5.2 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 (23.12.6) 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.
23.12.5.3 Pool resource constructors and destructors

synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

1. Requires: upstream is the address of a valid memory resource.
2. 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.

3. 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();

4. Effects: Calls release().

23.12.5.4 Pool resource members

void release();

1. 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;

2. Returns: The value of the upstream argument provided to the constructor of this object.

pool_options options() const;

3. 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;

4. Returns: A pointer to allocated storage (6.6.4.4.1) 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 (23.12).

5. 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().

6. Throws: Nothing unless upstream_resource()->allocate() throws.

void do_deallocate(void* p, size_t bytes, size_t alignment) override;

7. 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().

8. Throws: Nothing.

bool synchronized_pool_resource::do_is_equal(
    const memory_resource& other) const noexcept override;

9. Returns: this == dynamic_cast<const synchronized_pool_resource*>(&other).

bool unsynchronized_pool_resource::do_is_equal(
    const memory_resource& other) const noexcept override;

10. Returns: this == dynamic_cast<const unsynchronized_pool_resource*>(&other).
23.12.6 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()) {}
    explicit monotonic_buffer_resource(size_t initial_size)
      : monotonic_buffer_resource(initial_size, get_default_resource()) {}
    monotonic_buffer_resource(void* buffer, size_t buffer_size)
      : monotonic_buffer_resource(buffer, buffer_size, get_default_resource()) {}
    
    monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;
    virtual ~monotonic_buffer_resource();
    monotonic_buffer_resource& operator=(const monotonic_buffer_resource&) = delete;
  
    void release();
    memory_resource* upstream_resource() const;
  
  protected:
    void* do_allocate(size_t bytes, size_t alignment) override;
    void do_deallocate(void* p, size_t bytes, size_t alignment) override;
    bool do_is_equal(const memory_resource& other) const noexcept override;
  
  }
}

23.12.6.1 monotonic_buffer_resource constructor 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_rscc 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.
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).

~monotonic_buffer_resource();

Effects: Calls release().

23.12.6.2 monotonic_buffer_resource members

void release();

Effects: Calls upstream_rsrc->deallocate() as necessary to release all allocated memory.

[Note: The memory is released back to upstream_rsrc even if some blocks that were allocated from this have not been deallocated from this. — end note]

memory_resource* upstream_resource() const;

Returns: The value of upstream_rsrc.

void* do_allocate(size_t bytes, size_t alignment) override;

Returns: A pointer to allocated storage (6.6.4.4.1) 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 (23.12).

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.

void do_deallocate(void* p, size_t bytes, size_t alignment) override;

Effects: None.

Throws: Nothing.

Remarks: Memory used by this resource increases monotonically until its destruction.

bool do_is_equal(const memory_resource& other) const noexcept override;

Returns: this == dynamic_cast<const monotonic_buffer_resource*>(&other).

23.13 Class template scoped_allocator_adaptor

23.13.1 Header <scoped_allocator> synopsis

namespace std {

// class template scoped_allocator_adaptor

template<class OuterAlloc, class... InnerAllocs>
class scoped_allocator_adaptor;

// 23.13.5, 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 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. \[\text{Note: The \texttt{scoped\_allocator\_adaptor} is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions.} \]

```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<typename OuterTraits::template rebind_alloc<Tp>, InnerAllocs...>;
            };
            scoped_allocator_adaptor();
            template<class OuterA2>
            scoped_allocator_adaptor(OuterA2&& outerAlloc, const InnerAllocs&... innerAllocs) noexcept;
            scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& other) noexcept;
            scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;
            scoped_allocator_adaptor& operator=(const scoped_allocator_adaptor&) = default;
            scoped_allocator_adaptor& operator=(scoped_allocator_adaptor&&) = default;
            ~scoped_allocator_adaptor();
            inner_allocator_type& inner_allocator() noexcept;
            const inner_allocator_type& inner_allocator() const noexcept;
            outer_allocator_type& outer_allocator() noexcept;
            const outer_allocator_type& outer_allocator() const noexcept;
    }
}
```

§ 23.13.1
[[nodiscard]] pointer allocate(size_type n);
[[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);
void deallocate(pointer p, size_type n);
size_type max_size() const;

template<class T, class... Args>
  void construct(T* p, Args&&... args);

```cpp
using inner_allocator_type = see below;
```

```cpp
1 Type: scoped_allocator_adaptor<OuterAlloc> if sizeof...(InnerAllocs) is zero; otherwise, scoped_allocator_adaptor<InnerAllocs>.
```

```cpp
using propagate_on_container_copy_assignment = see below;
```

```cpp
2 Type: true_type if allocator_traits<A>::propagate_on_container_copy_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.
```

```cpp
using propagate_on_container_move_assignment = see below;
```

```cpp
3 Type: true_type if allocator_traits<A>::propagate_on_container_move_assignment::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.
```

```cpp
using propagate_on_container_swap = see below;
```

```cpp
4 Type: true_type if allocator_traits<A>::propagate_on_container_swap::value is true for any A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.
```

```cpp
using is_always_equal = see below;
```

```cpp
5 Type: true_type if allocator_traits<A>::is_always_equal::value is true for every A in the set of OuterAlloc and InnerAllocs...; otherwise, false_type.
```

### 23.13.3 Scoped allocator adaptor constructors

```cpp
scoped_allocator_adaptor());
```

```cpp
1 Effects: Value-initializes the OuterAlloc base class and the inner allocator object.
```

```cpp
template<class OuterA2>
  scoped_allocator_adaptor(OuterA2&& outerAlloc, const InnerAllocs&... innerAllocs) noexcept;
```

```cpp
2 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 adaptor from the argument list).
```
Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v<OuterAlloc, OuterA2>` is true.

```cpp
scoped_allocator_adaptor(const scoped_allocator_adaptor& other) noexcept;
```

**Effects:** Initializes each allocator within the adaptor with the corresponding allocator from `other`.

```cpp
scoped_allocator_adaptor(scoped_allocator_adaptor&& other) noexcept;
```

**Effects:** Move constructs each allocator within the adaptor with the corresponding allocator from `other`.

```cpp
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`.

```cpp
template<class OuterA2>
scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, InnerAllocs...>&& other) noexcept;
```

**Effects:** Initializes each allocator within the adaptor with the corresponding allocator rvalue from `other`.

Remarks: This constructor shall not participate in overload resolution unless `is_constructible_v<OuterAlloc, const OuterA2&>` is true.

23.13.4 Scoped allocator adaptor members

In the construct member functions, `OUTERMOST(x)` is `x` if `x` does not have an `outer_allocator()` member function and `OUTERMOST(x.outer_allocator())` otherwise; `OUTERMOST_ALLOC_TRAITS(x)` is `allocator_traits<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]

```cpp
inner_allocator_type& inner_allocator() noexcept;
```

**Returns:** `*this` if `sizeof...(InnerAllocs)` is zero; otherwise, `inner`.

```cpp
const inner_allocator_type& inner_allocator() const noexcept;
```

**Returns:** `static_cast<OuterAlloc&>(*this)`.

```cpp
outer_allocator_type& outer_allocator() noexcept;
```

**Returns:** `static_cast<OuterAlloc&>(*this)`.

```cpp
[[nodiscard]] pointer allocate(size_type n);
```

**Returns:** `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n)`.

```cpp
[[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);
```

**Returns:** `allocator_traits<OuterAlloc>::allocate(outer_allocator(), n, hint)`.

```cpp
void deallocate(pointer p, size_type n) noexcept;
```

**Effects:** As if by: `allocator_traits<OuterAlloc>::deallocate(outer_allocator(), p, n)`.

```cpp
size_type max_size() const;
```

**Returns:** `allocator_traits<OuterAlloc>::max_size(outer_allocator())`.

```cpp
template<class T, class... Args>
void construct(T* p, Args&&... args);
```

**Effects:**

(9.1) If `uses_allocator_v<T, inner_allocator_type>` is false and `is_constructible_v<T, Args...>` is true, calls:
OUTERMOST_ALLOC_TRAITS(*this)::construct(
  OUTERMOST(*this), p, std::forward<Args>(args)...)  
(9.2)  
— Otherwise, if uses_allocator_v<T, inner_allocator_type> is true
  and is_constructible_v<T, allocator_arg_t, inner_allocator_type&, Args...> is true, calls:
  OUTERMOST_ALLOC_TRAITS(*this)::construct(
    OUTERMOST(*this), p, allocator_arg, inner_allocator(), std::forward<Args>(args)...)  
(9.3)  
— Otherwise, if uses_allocator_v<T, inner_allocator_type> is true
  and is_constructible_v<T, Args..., inner_allocator_type&> is true, calls:
  OUTERMOST_ALLOC_TRAITS(*this)::construct(
    OUTERMOST(*this), p, std::forward<Args>(args)..., inner_allocator())  
(9.4)  
— Otherwise, the program is ill-formed. [Note: An error will result if uses_allocator evaluates to true but the specific constructor does not take an allocator. This definition prevents a silent failure to pass an inner allocator to a contained element. —end note]

template<class T1, class T2, class... Args1, class... Args2>
void construct(pair<T1, T2>* p, piecewise_construct_t, tuple<Args1...> x, tuple<Args2...> y);

Requires: All of the types in Args1 and Args2 shall be CopyConstructible (Table 24).

Effects: Constructs a tuple object xprime from x by the following rules:

(11.1) — If uses_allocator_v<T1, inner_allocator_type> is false and is_constructible_v<T1, Args1...> is true, then xprime is:
  tuple_cat(
    tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator()),
    std::move(x))

(11.2) — Otherwise, if uses_allocator_v<T1, inner_allocator_type> is true and is_constructible_v<T1, allocator_arg_t, inner_allocator_type&, Args1...> is true, then xprime is:
  tuple_cat(
    tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator()),
    std::move(x))

(11.3) — Otherwise, if uses_allocator_v<T1, inner_allocator_type> is true and is_constructible_v<T1, Args1..., inner_allocator_type&> is true, then xprime is:
  tuple_cat(std::move(x), tuple<inner_allocator_type&>(inner_allocator()))

(11.4) — Otherwise, the program is ill-formed.

and constructs a tuple object yprime from y by the following rules:

(11.5) — If uses_allocator_v<T2, inner_allocator_type> is false and is_constructible_v<T2, Args2...> is true, then yprime is:
  tuple_cat(
    tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator()),
    std::move(y))

(11.6) — Otherwise, if uses_allocator_v<T2, inner_allocator_type> is true and is_constructible_v<T2, allocator_arg_t, inner_allocator_type&, Args2...> is true, then yprime is:
  tuple_cat(
    tuple<allocator_arg_t, inner_allocator_type&>(allocator_arg, inner_allocator()),
    std::move(y))

(11.7) — Otherwise, if uses_allocator_v<T2, inner_allocator_type> is true and is_constructible_v<T2, Args2..., inner_allocator_type&> is true, then yprime is:
  tuple_cat(std::move(y), tuple<inner_allocator_type&>(inner_allocator()))

(11.8) — Otherwise, the program is ill-formed.

then calls:

OUTERMOST_ALLOC_TRAITS(*this)::construct(
  OUTERMOST(*this), p, piecewise_construct, std::move(xprime), std::move(yprime))

template<class T1, class T2>
void construct(pair<T1, T2>* p);

Effects: Equivalent to:

  construct(p, piecewise_construct, tuple<>(), tuple<>());
template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, U&& x, V&& y);

Effects: Equivalent to:
const p, piecewise_construct,
forward_as_tuple((std::forward<U>(x)),
forward_as_tuple((std::forward<V>(y)));

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, const pair<U, V>& x);

Effects: Equivalent to:
const p, piecewise_construct,
forward_as_tuple(x.first),
forward_as_tuple(x.second);

template<class T1, class T2, class U, class V>
void construct(pair<T1, T2>* p, pair<U, V>&& x);

Effects: Equivalent to:
const p, piecewise_construct,
forward_as_tuple((std::forward<U>(x.first)),
forward_as_tuple((std::forward<V>(x.second)));

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.

23.13.5 Scoped allocator operators

template<class OuterA1, class OuterA2, class... InnerAllocs>
bool operator==(const scoped_allocator_adaptor<OuterA1, InnerAllocs...>& a,
const scoped_allocator_adaptor<OuterA2, InnerAllocs...>& b) noexcept;

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).

23.14 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 (8.5.1.2, 16.3.1.1). 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 28), 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.

Such a type is a function pointer or a class type which has a member operator() or a class type which has a conversion to a pointer to function.
23.14.1 Header <functional> synopsis

namespace std {
  // 23.14.4, invoke
  template<class F, class... Args>
  invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
    noexcept(is_nothrow_invocable_v<F, Args...>);

  // 23.14.5, 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;

  // 23.14.6, arithmetic operations
  template<class T = void> struct plus;
  template<class T = void> struct minus;
  template<class T = void> struct multiplies;
  template<class T = void> struct divides;
  template<class T = void> struct modulus;
  template<class T = void> struct negate;
  template<> struct plus<void>;
  template<> struct minus<void>;
  template<> struct multiplies<void>;
  template<> struct divides<void>;
  template<> struct modulus<void>;
  template<> struct negate<void>;

  // 23.14.7, 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>;

  // 23.14.8, 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>;

  // 23.14.9, 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>;
}
// 23.14.10, function template not_fn
template<class F> unspecified not_fn(F&& f);

// 23.14.11, 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;
    .
    .
    see below _M;
}

// 23.14.12, member function adaptors
template<class R, class T>
unspecified mem_fn(R T::* noexcept;

// 23.14.13, polymorphic function wrappers
class bad_function_call;

template<class R, class... ArgTypes>
class function<R(ArgTypes...)>

template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&) noexcept;

template<class R, class... ArgTypes>
bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

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;

// 23.14.14, searchers

template<class ForwardIterator, class BinaryPredicate = equal_to<>>
class default_searcher;

template<class RandomAccessIterator,
class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
class BinaryPredicate = equal_to<>>
class boyer_moore_searcher;

template<class RandomAccessIterator,
class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
class BinaryPredicate = equal_to<>>
class boyer_moore_horspool_searcher;

// 23.14.15, hash function primary template

template<class T>
struct hash;
// 23.14.11, 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;

1 [Example: If a C++ program wants to have a by-element addition of two vectors a and b containing double and put the result into a, it can do:
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
—end example]

2 [Example: To negate every element of a:
transform(a.begin(), a.end(), a.begin(), negate<double>());
—end example]

23.14.2 Definitions [func.def]
1 The following definitions apply to this Clause:
2 A call signature is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.
3 A callable type is a function object type (23.14) or a pointer to member.
4 A callable object is an object of a callable type.
5 A call wrapper type is a type that holds a callable object and supports a call operation that forwards to that object.
6 A call wrapper is an object of a call wrapper type.
7 A target object is the callable object held by a call wrapper.

23.14.3 Requirements [func.require]
1 Define \texttt{INVOKE}(f, t_1, t_2, \ldots, t_N) as follows:
\begin{enumerate}
\item[(1.1)] \texttt{(t}_1.\ast f)(t_2, \ldots, t_N) \text{ when } f \text{ is a pointer to a member function of a class } T \text{ and } \text{is_base_of}_{\cdot v<T, \text{remove_reference}_{\cdot v<\text{decltype}(t_1)}} \text{ is true;}
\item[(1.2)] \texttt{(t}_1.\text{get().}\ast f)(t_2, \ldots, t_N) \text{ when } f \text{ is a pointer to a member function of a class } T \text{ and } \text{remove_cvref}_{\cdot v<\text{decltype}(t_1)} \text{ is a specialization of } \text{reference}_{\cdot wrapper};
\item[(1.3)] \texttt{(*t}_1.\ast f)(t_2, \ldots, t_N) \text{ when } f \text{ is a pointer to a member function of a class } T \text{ and } t_1 \text{ does not satisfy the previous two items;}
\item[(1.4)] \texttt{t}_1.\ast f \text{ when } N = 1 \text{ and } f \text{ is a pointer to data member of a class } T \text{ and } \text{is_base_of}_{\cdot v<T, \text{remove_reference}_{\cdot v<\text{decltype}(t_1)}} \text{ is true;}
\item[(1.5)] \texttt{t}_1.\text{get().}\ast f \text{ when } N = 1 \text{ and } f \text{ is a pointer to data member of a class } T \text{ and } \text{remove_cvref}_{\cdot v<\text{decltype}(t_1)} \text{ is a specialization of } \text{reference}_{\cdot wrapper};
\item[(1.6)] \texttt{(*t}_1.\ast f \text{ when } N = 1 \text{ and } f \text{ is a pointer to data member of a class } T \text{ and } t_1 \text{ does not satisfy the previous two items;}
\item[(1.7)] \texttt{f(t}_1, t_2, \ldots, t_N) \text{ in all other cases.}
\end{enumerate}
2 Define \texttt{INVOKE<R>(f, t_1, t_2, \ldots, t_N)} as \texttt{static_cast\langle void}(\texttt{INVOKE(f, t_1, t_2, \ldots, t_N)) \text{ if } R \text{ is cv void, otherwise } \texttt{INVOKE(f, t_1, t_2, \ldots, t_N)} \text{ implicitly converted to } R.\]
3 Every call wrapper (23.14.2) shall be 	exttt{MoveConstructible}. A forwarding call wrapper is a call wrapper that can be called with an arbitrary argument list and delivers the arguments to the wrapped callable object as references. This forwarding step shall ensure that rvalue arguments are delivered as rvalue references and lvalue arguments are delivered as lvalue references. A \textit{simple call wrapper} is a forwarding call wrapper that is \texttt{CopyConstructible} and \texttt{CopyAssignable} and whose copy constructor, move constructor, copy assignment operator, and move assignment operator do not throw exceptions. [Note: In a typical implementation forwarding call wrappers have an overloaded function call operator of the form
\begin{verbatim}
\text{template<class... UnBoundArgs>
  R operator()\langle UnBoundArgs&&\ldots\ undbound_args\rangle cv-qual;
\end{verbatim}]

§ 23.14.3
23.14.4 Function template invoke

template<class F, class... Args>
invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
    noexcept(is_nothrow_invocable_v<F, Args...>);

1 Returns: INVOKE(std::forward<F>(f), std::forward<Args>(args)...) (23.14.3).

23.14.5 Class template reference_wrapper

namespace std {
    template<class T> class reference_wrapper {
        public:
            // types
            using type = T;

            // construct/copy/destroy
            template<class U>
                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 CopyConstructible and CopyAssignable wrapper around a reference to an
object or function of type T.

2 reference_wrapper<T> shall be a trivially copyable type (6.7).

23.14.5.1 reference_wrapper construct/copy/destroy

template<class U>
    reference_wrapper(U&& u) noexcept(see below);

1 Remarks: Let FUN denote the exposition-only functions
        void FUN(T&) noexcept;
        void FUN(T&&) = delete;
    This constructor shall not participate in overload resolution unless the expression FUN(declval<U>())
is well-formed and is_same_v<decay_t<T&>, reference_wrapper> is false. The expression inside
    noexcept is equivalent to noexcept(FUN(declval<U>())).

2 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;

3 Effects: Constructs a reference_wrapper object that stores a reference to x.get().

23.14.5.2 reference_wrapper assignment

reference_wrapper& operator=(const reference_wrapper& x) noexcept;

1 Postconditions: *this stores a reference to x.get().
23.14.5.3 reference_wrapper access

operator T& () const noexcept;

Returns: The stored reference.

T& get() const noexcept;

Returns: The stored reference.

23.14.5.4 reference_wrapper invocation

template<class... ArgTypes>
invoke_result_t<T&, ArgTypes...>
operator() (ArgTypes&&... args) const;

Returns: INVOKE(get(), std::forward<ArgTypes>(args)...). (23.14.3)

23.14.5.5 reference_wrapper helper functions

template<class T> reference_wrapper<T> ref(T& t) noexcept;

Returns: reference_wrapper<T>(t).

template<class T> reference_wrapper<T> ref(reference_wrapper<T> t) noexcept;

Returns: ref(t.get()).

template<class T> reference_wrapper<const T> cref(const T& t) noexcept;

Returns: reference_wrapper<const T>(t).

template<class T> reference_wrapper<const T> cref(reference_wrapper<T> t) noexcept;

Returns: cref(t.get()).

23.14.6 Arithmetic operations

The library provides basic function object classes for all of the arithmetic operators in the language (8.5.5, 8.5.6).

23.14.6.1 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).

23.14.6.2 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));

23.14.6.3 Class template multiplies
[arithmetic.operations.multiplies]

template<class T = void> struct multiplies {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;

1 Returns: x * y.

template<> struct multiplies<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) * std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
  -> decltype(std::forward<T>(t) * std::forward<U>(u));

2 Returns: std::forward<T>(t) * std::forward<U>(u).

23.14.6.4 Class template divides
[arithmetic.operations.divides]

template<class T = void> struct divides {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;

1 Returns: x / y.

template<> struct divides<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) / std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
  -> decltype(std::forward<T>(t) / std::forward<U>(u));

2 Returns: std::forward<T>(t) / std::forward<U>(u).

23.14.6.5 Class template modulus
[arithmetic.operations.modulus]

template<class T = void> struct modulus {
  constexpr T operator()(const T& x, const T& y) const;
};

constexpr T operator()(const T& x, const T& y) const;

1 Returns: x % y.

template<> struct modulus<void> {
  template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
    -> decltype(std::forward<T>(t) % std::forward<U>(u));

§ 23.14.6.5 599
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));

23.14.6.6 Class template negate

`template<class T = void> struct negate {
    constexpr T operator()(const T& x) const;
};`

`constexpr T operator()(const T& x) const;`

1 Returns: -x.

template<> struct negate<void> {
    template<class T> constexpr auto operator()(T&& t) const
       -> decltype(-std::forward<T>(t));

    using is_transparent = unspecified;
};

template<class T> constexpr auto operator()(T&& t) const
   -> decltype(-std::forward<T>(t));

2 Returns: -std::forward<T>(t).

23.14.7 Comparisons

The library provides basic function object classes for all of the comparison operators in the language (8.5.9, 8.5.10).

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.

23.14.7.1 Class template equal_to

`template<class T = void> struct equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
};`

`constexpr bool operator()(const T& x, const T& y) const;`

1 Returns: x == y.

template<> struct equal_to<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
       -> decltype(std::forward<T>(t) == std::forward<U>(u));

    using is_transparent = unspecified;
};

template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
   -> decltype(std::forward<T>(t) == std::forward<U>(u));

2 Returns: std::forward<T>(t) == std::forward<U>(u).
23.14.7.2 Class template not_equal_to

```cpp
template<class T = void> struct not_equal_to {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

1. Returns: \( x \neq y \).

```cpp
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));
}
```

2. Returns: \( \text{std::forward}(t) \neq \text{std::forward}(u) \).

23.14.7.3 Class template greater

```cpp
template<class T = void> struct greater {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

1. Returns: \( x > y \).

```cpp
template<> struct greater<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) > std::forward<U>(u));
}
```

2. Returns: \( \text{std::forward}(t) > \text{std::forward}(u) \).

23.14.7.4 Class template less

```cpp
template<class T = void> struct less {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

1. Returns: \( x < y \).

```cpp
template<> struct less<void> {
    template<class T, class U> constexpr auto operator()(T&& t, U&& u) const
        -> decltype(std::forward<T>(t) < std::forward<U>(u));
}
```

2. Returns: \( \text{std::forward}(t) < \text{std::forward}(u) \).

23.14.7.5 Class template greater_equal

```cpp
template<class T = void> struct greater_equal {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

1. 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 \text{std::forward}(u));
}
```

2. Returns: \( \text{std::forward}(t) \geq \text{std::forward}(u) \).
constexpr bool operator()(const T& x, const T& y) const;

1 Returns: x >= y.

template<>
struct greater_equal<void> {

template<class T, class U>
constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) >= std::forward<U>(u));

using is_transparent = unspecified;
};

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).

23.14.7.6 Class template less_equal

template<class T = void>
struct less_equal {

constexpr bool operator()(const T& x, const T& y) const;

};

constexpr bool operator()(const T& x, const T& y) const;

1 Returns: x <= y.

template<>
struct less_equal<void> {

template<class T, class U>
constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) <= std::forward<U>(u));

using is_transparent = unspecified;
};

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).

23.14.8 Logical operations

1 The library provides basic function object classes for all of the logical operators in the language (8.5.14, 8.5.15, 8.5.2.1).

23.14.8.1 Class template logical_and

template<class T = void>
struct logical_and {

constexpr bool operator()(const T& x, const T& y) const;

};

constexpr bool operator()(const T& x, const T& y) const;

1 Returns: x && y.

template<>
struct logical_and<void> {

template<class T, class U>
constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) && std::forward<U>(u));

using is_transparent = unspecified;
};

template<class T, class U>
constexpr auto operator()(T&& t, U&& u) const
-> decltype(std::forward<T>(t) && std::forward<U>(u));

2 Returns: std::forward<T>(t) && std::forward<U>(u).
23.14.8.2 Class template logical_or

```cpp
template<class T = void> struct logical_or {
    constexpr bool operator()(const T& x, const T& y) const;
};
```

1 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;
};
```

2 Returns: \( \text{std::forward}(t) \| \text{std::forward}(u) \).

23.14.8.3 Class template logical_not

```cpp
template<class T = void> struct logical_not {
    constexpr bool operator()(const T& x) const;
};
```

1 Returns: \( \neg x \).

```cpp
template<> struct logical_not<void> {
    template<class T> constexpr auto operator()(T&& t) const
        -> decltype(!std::forward<T>(t));

    using is_transparent = unspecified;
};
```

2 Returns: \( \neg \text{std::forward}(t) \).

23.14.9 Bitwise operations

1 The library provides basic function object classes for all of the bitwise operators in the language (8.5.11, 8.5.13, 8.5.12, 8.5.2.1).

23.14.9.1 Class template bit_and

```cpp
template<class T = void> struct bit_and {
    constexpr T operator()(const T& x, const T& y) const;
};
```

1 Returns: \( x \& y \).

```cpp
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;
};
```

2 Returns: \( \text{std::forward}(t) \& \text{std::forward}(u) \).
23.14.9.2 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).
```

23.14.9.3 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).
```

23.14.9.4 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).
```
23.14.10 Function template not_fn

template<class F> unspecified not_fn(F&& f);

Effects: Equivalent to: return call_wrapper(std::forward<F>(f)); where call_wrapper is an exposition only class defined as follows:

class call_wrapper {
    using FD = decay_t<F>;
    FD fd;

    explicit call_wrapper(F&& f);
}

Effects: Equivalent to:
return !INVOKE(fd, std::forward<Args>(args)...);

template<class... Args>
auto operator()(Args&&...) &
-> decltype(!declval<invoke_result_t<FD&, Args...>>());

Effects: Equivalent to:
return !INVOKE(std::move(fd), std::forward<Args>(args)...);

23.14.11 Function object binders

This subclause describes a uniform mechanism for binding arguments of callable objects.

§ 23.14.11 605
23.14.11.1 Class template is_bind_expression

```cpp
namespace std {
    template<class T> struct is_bind_expression;  // see below
}
```

1. The class template `is_bind_expression` can be used to detect function objects generated by `bind`. The function template `bind` uses `is_bind_expression` to detect subexpressions.

2. Instantiations of the `is_bind_expression` template shall meet the `UnaryTypeTrait` requirements (23.15.1). 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 user-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.

23.14.11.2 Class template is_placeholder

```cpp
namespace std {
    template<class T> struct is_placeholder;  // see below
}
```

1. 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.

2. Instantiations of the `is_placeholder` template shall meet the `UnaryTypeTrait` requirements (23.15.1). 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 user-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.

23.14.11.3 Function template bind

```cpp
template<class F, class... BoundArgs>
unspecified bind(F&& f, BoundArgs&&... bound_args);
```

1. In the text that follows:

   1.1. `FD` is the type `decay_t<F>`,
   1.2. `fd` is an lvalue of type `FD` constructed from `std::forward<F>(f)`,
   1.3. `T_i` is the `i`th type in the template parameter pack `BoundArgs`,
   1.4. `TD_i` is the type `decay_t<T_i>`,
   1.5. `t_i` is the `i`th argument in the function parameter pack `bound_args`,
   1.6. `td_i` is an lvalue of type `TD_i` constructed from `std::forward<T_i>(t_i)`,
   1.7. `U_j` is the `j`th deduced type of the `UnBoundArgs&&...` parameter of the forwarding call wrapper, and
   1.8. `u_j` is the `j`th argument associated with `U_j`.

2. Requires: `is_constructible_v<FD, F>` shall be true. For each `T_i` in `BoundArgs`, `is_constructible_v<FD, T_i>` shall be true. `INVOKEx(fd, w_1, w_2, ..., w_N)` (23.14.3) shall be a valid expression for some values `w_1, w_2, ..., w_N`, where `N` has the value `sizeof...(bound_args)`. The cv-qualifiers `cv` of the call wrapper `g`, as specified below, shall be neither `volatile` nor `const volatile`.

3. Returns: A forwarding call wrapper `g` (23.14.3). The effect of `g(u_1, u_2, ..., u_M)` shall be

   `INVOKEx(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 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.

4. Throws: Nothing unless the construction of `fd` or of one of the values `td_i` throws an exception.

5. Remarks: The return type shall satisfy the requirements of `MoveConstructible`. If all of `FD` and `TD_i` satisfy the requirements of `CopyConstructible`, then the return type shall satisfy the requirements of `CopyConstructible`. [Note: This implies that all of `FD` and `TD_i` are `MoveConstructible`. — end note]
template<class R, class F, class... BoundArgs>
unspec specified bind(F&& f, BoundArgs&&... bound_args);

Requires: is_constructible_v<FD, F> shall be true. For each Ti in BoundArgs, is_constructible_v<TD, Ti> shall be true. INVOKE(fd, w1, w2, ..., wN) shall be a valid expression for some values w1, w2, ..., wN, where N has the value sizeof...(bound_args). The cv-qualifiers cv of the call wrapper g, as specified below, shall be neither volatile nor const volatile.

Returns: A forwarding call wrapper g (23.14.3). The effect of g(u1, u2, ..., uM) shall be

INVOKE<R>(fd, std::forward<V1>(v1), std::forward<V2>(v2), ..., std::forward<VN>(vN))

where the values and types of the bound arguments v1, v2, ..., vN are determined as specified below.

The copy constructor and move constructor of the forwarding call wrapper shall throw an exception if and only if the corresponding constructor of FD or of any of the types TDi throws an exception.

Remarks: The return type shall satisfy the requirements of CopyConstructible. If all of FD and TDi satisfy the requirements of CopyConstructible, then the return type shall satisfy the requirements of CopyConstructible. [Note: This implies that all of FD and TDi are MoveConstructible. —end note]

The values of the bound arguments v1, v2, ..., vN and their corresponding types V1, V2, ..., VN depend on the types TDi derived from the call to bind and the cv-qualifiers cv of the call wrapper g as follows:

(10.1) — if TDi is reference_wrapper<T>, the argument is td.get() and its type Vi is T;

(10.2) — if the value of is_bind_expression_v<TDi> is true, the argument is td(std::forward<Uj>(uj)...) and its type Vi is invoke_result_t_t<TF, cv & Uj...>&&;

(10.3) — if the value j of is_placeholder_v<TDi> is not zero, the argument is std::forward<Uj>(uj) and its type Vi is Uj&&;

(10.4) — otherwise, the value is td, and its type Vi is TDi cv &.

### 23.14.11.4 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 DefaultConstructible and CopyConstructible, and their default constructors and copy/move constructors shall not throw exceptions. It is implementation-defined whether placeholder types are CopyAssignable. CopyAssignable placeholders' copy assignment operators shall not throw exceptions.

Placeholders should be defined as:

```
inline constexpr unspec specified _1();
```

If they are not, they shall be declared as:

```
extern unspec specified _1;
```

### 23.14.12 Function template mem_fn

```
template<class R, class T> unspec specified mem_fn(R T::* pm) noexcept;
```

Returns: A simple call wrapper (23.14.2) fn such that the expression fn(t, a2, ..., aN) is equivalent to INVOKE(pm, t, a2, ..., aN) (23.14.3).

### 23.14.13 Polymorphic function wrappers

This subclause describes a polymorphic wrapper class that encapsulates arbitrary callable objects.
23.14.13.1  Class bad_function_call

An exception of type bad_function_call is thrown by function::operator() (23.14.13.2.4) when the function wrapper object has no target.

```cpp
namespace std {

    class bad_function_call : public exception {

    public:

        // 23.14.13.1.1, constructor
        bad_function_call() noexcept;
    }
}
```

**23.14.13.1.1  bad_function_call constructor**

bad_function_call() noexcept;

*Effects:* Constructs a bad_function_call object.

*Postconditions:* `what()` returns an implementation-defined NTBS.

23.14.13.2  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;

            // 23.14.13.2.1, construct/copy/destroy
            function() noexcept;
            function(nullptr_t) noexcept;
            function(const function&);
            function(function&&);
            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();

            // 23.14.13.2.2, function modifiers
            void swap(function&) noexcept;

            // 23.14.13.2.3, function capacity
            explicit operator bool() const noexcept;

            // 23.14.13.2.4, function invocation
            R operator()(ArgTypes...) const;

            // 23.14.13.2.5, function target access
            const type_info& target_type() const noexcept;
            template<class T>
            T* target() const 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>;
    }
```
The function class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects (23.14.2), given a call signature (23.14.2), allowing functions to be first-class objects.

A callable type (23.14.2) F is Lvalue-Callable for argument types ArgTypes... and return type R if the expression \texttt{INVOKE}<R>(\texttt{declval}<F&>(), \texttt{declval}<ArgTypes>()), considered as an unevaluated operand (8.2), is well-formed (23.14.3). The function class template is a call wrapper (23.14.2) whose call signature (23.14.2) is R(ArgTypes...).

[Note: The types deduced by the deduction guides for function may change in future versions of this International Standard. —end note]

23.14.13.2.1 function construct/copy/destroy

\begin{verbatim}
function() noexcept;
\end{verbatim}

\textit{Postconditions: !*this.}

\begin{verbatim}
function(nullptr_t) noexcept;
\end{verbatim}

\textit{Postconditions: !this.}

\begin{verbatim}
function(const function& f);
\end{verbatim}

\textit{Postconditions: !this if !f; otherwise, *this targets a copy of f.target().}

\textit{Throws:} Shall not throw exceptions if f’s target is a specialization of reference_wrapper or a function pointer. Otherwise, may throw bad_alloc or any exception thrown by the copy constructor of the stored callable object. [Note: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where f’s target is an object holding only a pointer or reference to an object and a member function pointer. —end note]

\begin{verbatim}
function(function&& f);
\end{verbatim}

\textit{Postconditions: If !f, *this has no target; otherwise, the target of *this is equivalent to the target of f before the construction, and f is in a valid state with an unspecified value.}

\textit{Throws:} Shall not throw exceptions if f’s target is a specialization of reference_wrapper or a function pointer. Otherwise, may throw bad_alloc or any exception thrown by the copy or move constructor of the stored callable object. [Note: Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where f’s target is an object holding only a pointer or reference to an object and a member function pointer. —end note]

\begin{verbatim}
template<class F> function(F f);
\end{verbatim}

\textit{Requires:} F shall be CopyConstructible.

\textit{Remarks:} This constructor template shall not participate in overload resolution unless F is Lvalue-Callable (23.14.13.2) for argument types ArgTypes... and return type R.

\textit{Postconditions: !this if any of the following hold:}
(9.1) \( f \) is a null function pointer value.
(9.2) \( f \) is a null member pointer value.
(9.3) \( F \) is an instance of the \texttt{function} class template, and \( f! \).

Otherwise, *this targets a copy of \( f \) initialized with \texttt{std::move(f)}. [\textit{Note:} Implementations should avoid the use of dynamically allocated memory for small callable objects, for example, where \( f \) is an object holding only a pointer or reference to an object and a member function pointer. —end note]\)

\textbf{Throws:} Shall not throw exceptions when \( f \) is a function pointer or a \texttt{reference_wrapper<T>} for some \( T \). Otherwise, may throw \texttt{bad_alloc} or any exception thrown by \( F \)'s copy or move constructor.

\begin{verbatim}
template<class F> function(F) -> function<see below>;
\end{verbatim}

\textit{Remarks:} This deduction guide participates in overload resolution only if \&\&\&.\texttt{operator()} is well-formed when treated as an unevaluated operand. In that case, if \texttt{decltype(&\&\&.\texttt{operator}())} is of the form \( R(G::*)(A...)\ cv k_{opt} \texttt{noexcept}_{opt} \) for a class type \( G \), then the deduced type is \texttt{function<R(A...)>.}

\begin{verbatim}
[Example:
void f() {
  int i{5};
  function g = [&](double) { return i; }; // deduces \texttt{function<int(double)>}
}
\end{verbatim}

—end example]

function& \texttt{operator=(const function& f)};

\textit{Effects:} As if by \texttt{function(f).swap(*this)};  
\textit{Returns:} \*this.

function& \texttt{operator=(function&& f)};

\textit{Effects:} Replaces the target of \*this with the target of \( f \).  
\textit{Returns:} \*this.

function& \texttt{operator=(nullptr_t) noexcept;}

\textit{Effects:} If \*this \(!=\) \texttt{nullptr}, destroys the target of \texttt{this}.  
\textit{Postconditions:} \(!(*\texttt{this}).\)

\textit{Returns:} \*this.

\texttt{template<class F> function& \texttt{operator=(F&& f)};}

\textit{Effects:} As if by: \texttt{function(std::forward<F>(f)).swap(*this)};  
\textit{Returns:} \*this.

\textit{Remarks:} This assignment operator shall not participate in overload resolution unless \texttt{decay_t<F>} is Lvalue-Callable (23.14.13.2) for argument types \texttt{ArgTypes}... and return type \( R \).

\begin{verbatim}
\texttt{template<class F> function& \texttt{operator=(reference_wrapper<F> f) noexcept;}
\end{verbatim}

\textit{Effects:} As if by: \texttt{function(f).swap(*this)};  
\textit{Returns:} \*this.

\texttt{~function();}

\textit{Effects:} If \*this \(!=\) \texttt{nullptr}, destroys the target of \texttt{this}.

23.14.13.2.2 \textit{function modifiers} [func.wrap.func.mod]

\texttt{void swap(function& other) noexcept;}

\textit{Effects:} Interchanges the targets of \*this and other.

23.14.13.2.3 \textit{function capacity} [func.wrap.func.cap]

\texttt{explicit operator bool() const noexcept;}

\textit{Returns:} \texttt{true} if \*this has a target, otherwise \texttt{false}.

\§ 23.14.13.2.3
23.14.13.2.4 function invocation

R operator()(ArgTypes... args) const;

1 Returns: \texttt{INVOKE}\langle R \rangle (f, \texttt{std::forward<ArgTypes}>(\texttt{args})...) (23.14.3), where \( f \) is the target object \((23.14.2)\) of \(*\texttt{this}\).

2 Throws: \texttt{bad_function_call} if \(!\texttt{!this}\); otherwise, any exception thrown by the wrapped callable object.

23.14.13.2.5 function target access

const type_info& target_type() const noexcept;

1 Returns: If \(*\texttt{this}\) has a target of type \(T\), \texttt{typeid(T)}; otherwise, \texttt{typeid(void)}.

template<class T> T* target() noexcept;
template<class T> const T* target() const noexcept;

2 Returns: If \(\texttt{target_type()} == \texttt{typeid(T)}\) a pointer to the stored function target; otherwise a null pointer.

23.14.13.2.6 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;

1 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;

2 Returns: (\texttt{bool})\(f\).

23.14.13.2.7 specialized algorithms

template<class R, class... ArgTypes>
void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2) noexcept;

1 Effects: As if by: \(f1.\texttt{swap}(f2)\);

23.14.14 Searchers

This subclause provides function object types (23.14) for operations that search for a sequence \([\texttt{pat}\_\texttt{first}, \texttt{pat}\_\texttt{last})\) in another sequence \([\texttt{first}, \texttt{last})\) 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 23.14.14 shall meet the \texttt{CopyConstructible} and \texttt{CopyAssignable} requirements. Template parameters named

\begin{itemize}
  \item \texttt{ForwardIterator},
  \item \texttt{ForwardIterator1},
  \item \texttt{ForwardIterator2},
  \item \texttt{RandomAccessIterator},
  \item \texttt{RandomAccessIterator1},
  \item \texttt{RandomAccessIterator2}, and
  \item \texttt{BinaryPredicate}
\end{itemize}

of templates specified in this subclause 23.14.14 shall meet the same requirements and semantics as specified in 28.1. Template parameters named \texttt{Hash} shall meet the requirements as specified in 20.5.3.4.

3 The Boyer-Moore searcher implements the Boyer-Moore search algorithm. The Boyer-Moore-Horspool searcher implements the Boyer-Moore-Horspool search algorithm. In general, the Boyer-Moore searcher will use more memory and give better runtime performance than Boyer-Moore-Horspool.
23.14.14.1 Class template default_searcher

```
template<class ForwardIterator1, class BinaryPredicate = equal_to<>>
class default_searcher {
  public:
    default_searcher(ForwardIterator1 pat_first, ForwardIterator1 pat_last,
                     BinaryPredicate pred = BinaryPredicate());

template<class ForwardIterator2>
  pair<ForwardIterator2, ForwardIterator2>
  operator()(ForwardIterator2 first, ForwardIterator2 last) const;

  private:
    ForwardIterator1 pat_first_; // exposition only
    ForwardIterator1 pat_last_;  // exposition only
    BinaryPredicate pred_;      // exposition only
};
```

```
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.
```

23.14.14.2 Class template boyer_moore_searcher

```
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 meet the DefaultConstructible requirements, the CopyConstructible requirements, and the CopyAssignable 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

1. `i` is the first iterator in the range `[first, last - (pat_last_ - pat_first_))` such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds: `pred(*(i + n), *(pat_first_ + n)) != false`, and
2. `j == next(i, distance(pat_first_, pat_last_))`.

Returns `make_pair(first, first)` if `[pat_first_, pat_last_)` is empty, otherwise returns `make_pair(last, last)` if no such iterator is found.

Complexity: At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

### 23.14.14.3 Class template `boyer_moore_horspool_searcher`

```
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());

private:
  RandomAccessIterator1 pat_first_; // exposition only
  RandomAccessIterator1 pat_last_; // exposition only
  Hash hash_; // exposition only
  BinaryPredicate pred_; // exposition only
};
```

`boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first,
                                RandomAccessIterator1 pat_last,
                                Hash hf = Hash(),
                                BinaryPredicate pred = BinaryPredicate());`

Requires: The value type of `RandomAccessIterator1` shall meet the `DefaultConstructible`, `CopyConstructible`, and `CopyAssignable` 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_horspool_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

\[ i \text{ is the first iterator in the range } [\text{first}, \text{last} - (\text{pat_last_} - \text{pat_first_})) \text{ such that for every non-negative integer } n \text{ less than } \text{pat_last_} - \text{pat_first_} \text{ the following condition holds:} \]

\[ \text{pred}((i + n), *(\text{pat_first_} + n)) != \text{false}, \text{ and} \]

\[ j = \text{next}(i, \text{distance}(\text{pat_first_}, \text{pat_last_})). \]

Returns make_pair(first, first) if \((\text{pat_first_}, \text{pat_last_})\) is empty, otherwise returns make_pair\((\text{last}, \text{last})\) if no such iterator is found.

Complexity: At most \((\text{last} - \text{first}) \times (\text{pat_last_} - \text{pat_first_})\) applications of the predicate.

23.14.15 Class template hash

The unordered associative containers defined in 26.5 use specializations of the class template hash (23.14.1) as the default hash function.

Each specialization of hash is either enabled or disabled, as described below. [Note: Enabled specializations meet the requirements of Hash, and disabled specializations do not. —end note] Each header that declares the template hash provides enabled specializations of hash for nullptr_t and all cv-unqualified arithmetic, enumeration, and pointer types. For any type Key for which neither the library nor the user provides an explicit or partial specialization of the class template hash, hash<Key> is disabled.

If the library provides an explicit or partial specialization of hash<Key>, that specialization is enabled except as noted otherwise, and its member functions are noexcept except as noted otherwise.

If H is a disabled specialization of hash, these values are false: is_default_constructible_v<H>, is_copy_constructible_v<H>, is_move_constructible_v<H>, is_copy_assignable_v<H>, and is_move_assignable_v<H>. Disabled specializations of hash are not function object types (23.14). [Note: This means that the specialization of hash exists, but any attempts to use it as a Hash will be ill-formed. —end note]

An enabled specialization hash<Key> will:

1. satisfy the Hash requirements (20.5.3.4), with Key as the function call argument type, the DefaultConstructible requirements (Table 22), the CopyAssignable requirements (Table 26),
2. be swappable (20.5.3.2) for lvalues,
3. satisfy the requirement that if \(k1 == k2\) is true, \(\text{h}(k1) == \text{h}(k2)\) is also true, where \(\text{h}\) is an object of type hash<Key> and \(k1\) and \(k2\) are objects of type Key;
4. satisfy the requirement that the expression \(\text{h}(k)\), where \(\text{h}\) is an object of type hash<Key> and \(k\) is an object of type Key, shall not throw an exception unless hash<Key> is a user-defined specialization that depends on at least one user-defined type.

23.15 Metaprogramming and type traits

This subclause describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time. It includes type classification traits, type property inspection traits, and type transformations. The type classification traits describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs. The type property inspection traits allow important characteristics of types or of combinations of types to be inspected. The type transformations allow certain properties of types to be manipulated.

All functions specified in this subclause are signal-safe (21.11.4).

23.15.1 Requirements

A UnaryTypeTrait 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 DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template integral_constant (23.15.3),
with the arguments to the template integral_constant determined by the requirements for the particular property being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the UnaryTypeTrait.

A BinaryTypeTrait 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 DefaultConstructible, CopyConstructible, and publicly and unambiguously derived, directly or indirectly, from its base characteristic, which is a specialization of the template integral_constant (23.15.3), with the arguments to the template integral_constant determined by the requirements for the particular relationship being described. The member names of the base characteristic shall not be hidden and shall be unambiguously available in the BinaryTypeTrait.

A TransformationTrait 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.

23.15.2 Header <type_traits> synopsis

```
namespace std {
  // 23.15.3, 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>;

  // 23.15.4.1, 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;

  // 23.15.4.2, 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;

  // 23.15.4.3, 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;
```

§ 23.15.2 615
template<class T> struct is_signed;
template<class T> struct is_unsigned;

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_copyAssignable;
template<class T> struct is_moveAssignable;

template<class T, class U> struct is_swappable_with;
template<class T> struct is_swappable;

template<class T> struct is_destructible;

template<class T, class... Args> struct is_trivially_constructible;
template<class T> struct is_trivially_default_constructible;
template<class T> struct is_trivially_copy_constructible;
template<class T> struct is_trivially_move_constructible;

template<class T, class U> struct is_trivially_assignable;
template<class T> struct is_trivially_copyAssignable;
template<class T> struct is_trivially_moveAssignable;

template<class T, class... Args> struct is_trivially_swappable_with;
template<class T> struct is_trivially_swappable;

template<class T> struct is_trivially_destructible;

template<class T, class... Args> struct is_nothrow_constructible;
template<class T> struct is_nothrow_default_constructible;
template<class T> struct is_nothrow_copy_constructible;
template<class T> struct is_nothrow_move_constructible;

template<class T, class U> struct is_nothrow_assignable;
template<class T> struct is_nothrow_copyAssignable;
template<class T> struct is_nothrow_moveAssignable;

template<class T, class U> struct is_nothrow_swappable_with;
template<class T> struct is_nothrow_swappable;

template<class T> struct is_nothrow_destructible;

template<class T> struct has_virtualDestructor;
template<class T> struct has_unique_object_representations;

// 23.15.5, type property queries
template<class T> struct alignment_of;
template<class T> struct rank;
template<class T, unsigned I = 0> struct extent;

// 23.15.6, 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 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;

// 23.15.7.1, const-volatile modifications
template<class T> struct remove_const;
template<class T> struct remove_volatile;

§ 23.15.2
template<class T> struct remove_cv;
template<class T> struct add_const;
template<class T> struct add_volatile;
template<class T> struct add_cv;

template<class T>
  using remove_const_t = typename remove_const<T>::type;
template<class T>
  using remove_volatile_t = typename remove_volatile<T>::type;
template<class T>
  using remove_cv_t = typename remove_cv<T>::type;
template<class T>
  using add_const_t = typename add_const<T>::type;
template<class T>
  using add_volatile_t = typename add_volatile<T>::type;
template<class T>
  using add_cv_t = typename add_cv<T>::type;

// 23.15.7.2, reference modifications
template<class T> struct remove_reference;
template<class T> struct add_lvalue_reference;
template<class T> struct add_rvalue_reference;

template<class T>
  using remove_reference_t = typename remove_reference<T>::type;
template<class T>
  using add_lvalue_reference_t = typename add_lvalue_reference<T>::type;
template<class T>
  using add_rvalue_reference_t = typename add_rvalue_reference<T>::type;

// 23.15.7.3, sign modifications
template<class T> struct make_signed;
template<class T> struct make_unsigned;

template<class T>
  using make_signed_t = typename make_signed<T>::type;
template<class T>
  using make_unsigned_t = typename make_unsigned<T>::type;

// 23.15.7.4, array modifications
template<class T> struct remove_extent;
template<class T> struct remove_all_extents;

template<class T>
  using remove_extent_t = typename remove_extent<T>::type;
template<class T>
  using remove_all_extents_t = typename remove_all_extents<T>::type;

// 23.15.7.5, pointer modifications
template<class T> struct remove_pointer;
template<class T> struct add_pointer;

template<class T>
  using remove_pointer_t = typename remove_pointer<T>::type;
template<class T>
  using add_pointer_t = typename add_pointer<T>::type;

// 23.15.7.6, other transformations
template<
  size_t Len,
  size_t Align = default-alignment> // see 23.15.7.6
  struct aligned_storage;
template<
  size_t Len,, class... Types>
  struct aligned_union;
template<class T> struct remove_cvref;
template<class T> struct decay;
template<bool, class T = void>
  struct enable_if;

§ 23.15.2
template<\text{bool}, \text{class T}, \text{class F}> \text{struct conditional};
template<\text{class... T}> \text{struct common\_type};
template<\text{class T}> \text{struct underlying\_type};
template<\text{class Fn, class... ArgTypes}> \text{struct invoke\_result};

\text{template<\text{size\_t Len, size\_t Align = default\_alignment}> // see 23.15.7.6}
\text{using aligned\_storage\_t = \text{typename aligned\_storage<Len, Align>::type};}
\text{template<\text{size\_t Len, class... Types}>}
\text{using aligned\_union\_t = \text{typename aligned\_union<Len, Types...>::type};}
\text{template<\text{class T}>}
\text{using remove\_cvref\_t = \text{typename remove\_cvref<T>::type};}
\text{template<\text{class T}>}
\text{using decay\_t = \text{typename decay<T>::type};}
\text{template<\text{bool b, class T = void}>}
\text{using enable\_if\_t = \text{typename enable\_if<b, T>::type};}
\text{template<\text{bool b, class T, class F}>}
\text{using conditional\_t = \text{typename conditional<b, T, F>::type};}
\text{template<\text{class... T}>}
\text{using common\_type\_t = \text{typename common\_type<T...>::type};}
\text{template<\text{class T}>}
\text{using underlying\_type\_t = \text{typename underlying\_type<T>::type};}
\text{template<\text{class Fn, class... ArgTypes}>}
\text{using invoke\_result\_t = \text{typename invoke\_result<Fn, ArgTypes...>::type};}
\text{template<\text{class... T}>}
\text{using void\_t = void;}

// 23.15.8, logical operator traits
\text{template<\text{class... B}> \text{struct conjunction};}
\text{template<\text{class... B}> \text{struct disjunction};}
\text{template<\text{class B}> \text{struct negation};}

// 23.15.9, endian
\text{enum class endian \{}
\text{\hspace{1em}little = see below,}
\text{\hspace{1em}big = see below,}
\text{\hspace{1em}native = see below\}};

// 23.15.4.1, primary type categories
\text{template<\text{class T}>}
\text{inline constexpr bool is\_void\_v = is\_void<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_null\_pointer\_v = is\_null\_pointer<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_integral\_v = is\_integral<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_floating\_point\_v = is\_floating\_point<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_array\_v = is\_array<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_pointer\_v = is\_pointer<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_lvalue\_reference\_v = is\_lvalue\_reference<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_rvalue\_reference\_v = is\_rvalue\_reference<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_member\_object\_pointer\_v = is\_member\_object\_pointer<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_member\_function\_pointer\_v = is\_member\_function\_pointer<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_enum\_v = is\_enum<T>::\text{value};}
\text{template<\text{class T}>}
\text{inline constexpr bool is\_union\_v = is\_union<T>::\text{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;

// 23.15.4.2, 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;

// 23.15.4.3, 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, 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<T, U>::value;

template<class T>
inline constexpr bool is_nothrow_swappable_v = is_nothrow_swappable<T>::value;

template<class T>
inline constexpr bool is_nothrow_destructible_v = is_nothrow_destructible<T>::value;

template<class T>
inline constexpr bool has_virtual_destructor_v = has_virtual_destructor<T>::value;

template<class T>
inline constexpr bool has_unique_object_representations_v = has_unique_object_representations<T>::value;

// 23.15.5, 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;
template<typename T, typename U>
inline constexpr bool is_same_v = is_same<T, U>::value;

template<typename Base, typename Derived>
inline constexpr bool is_base_of_v = is_base_of<Base, Derived>::value;

template<typename From, typename To>
inline constexpr bool is_convertible_v = is_convertible<From, To>::value;

template<typename Fn, typename... ArgTypes>
inline constexpr bool is_invocable_v = is_invocable<Fn, ArgTypes...>::value;

template<typename R, typename Fn, typename... ArgTypes>
inline constexpr bool is_invocable_r_v = is_invocable_r<R, Fn, ArgTypes...>::value;

template<typename Fn, typename... ArgTypes>
inline constexpr bool is_nothrow_invocable_v = is_nothrow_invocable<Fn, ArgTypes...>::value;

template<typename R, typename Fn, typename... ArgTypes>
inline constexpr bool is_nothrow_invocable_r_v = is_nothrow_invocable_r<R, Fn, ArgTypes...>::value;

// 23.15.8, logical operator traits

namespace std {
    template<typename... B>
    inline constexpr bool conjunction_v = conjunction<B...>::value;
    template<typename... B>
    inline constexpr bool disjunction_v = disjunction<B...>::value;
    template<typename B>
    inline constexpr bool negation_v = negation<B>::value;
}

The behavior of a program that adds specializations for any of the templates defined in this subclause is undefined unless otherwise specified.

Unless otherwise specified, an incomplete type may be used to instantiate a template in this subclause.

23.15.3 Helper classes

    namespace std {
        template<typename 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; }
        }
    }

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.

23.15.4 Unary type traits

This subclause contains templates that may be used to query the properties of a type at compile time.

Each of these templates shall be a UnaryTypeTrait (23.15.1) with a base characteristic of true_type if the corresponding condition is true, otherwise false_type.

23.15.4.1 Primary type categories

The primary type categories correspond to the descriptions given in subclause 6.7 of the C++ standard.

For any given type T, the result of applying one of these templates to T and to cv T shall yield the same result.

[Note: For any given type T, exactly one of the primary type categories has a value member that evaluates to true. — end note]
Table 40 — Primary type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_void;</td>
<td>T is void</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_null_pointer;</td>
<td>T is nullptr_t (6.7.1)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_integral;</td>
<td>T is an integral type (6.7.1)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_floating_point;</td>
<td>T is a floating-point type (6.7.1)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_array;</td>
<td>T is an array type (6.7.2) of known or unknown extent</td>
<td>Class template array (26.3.7) is not an array type.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_pointer;</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>template&lt;class T&gt; struct is_lvalue_reference;</td>
<td>T is an lvalue reference type (11.3.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_rvalue_reference;</td>
<td>T is an rvalue reference type (11.3.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_object_pointer;</td>
<td>T is a pointer to data member</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_member_function_pointer;</td>
<td>T is a pointer to member function</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_enum;</td>
<td>T is an enumeration type (6.7.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_union;</td>
<td>T is a union type (6.7.2)</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_class;</td>
<td>T is a non-union class type (6.7.2)</td>
<td></td>
</tr>
</tbody>
</table>

23.15.4.2 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 41 — Composite type category predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; struct is_reference;</td>
<td>T is an lvalue reference or an rvalue reference</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_arithmetic;</td>
<td>T is an arithmetic type (6.7.1)</td>
<td></td>
</tr>
<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>
</tbody>
</table>

§ 23.15.4.2
Table 41 — 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;</td>
<td>T is a pointer-to-member type</td>
<td>(6.7.2)</td>
</tr>
<tr>
<td>struct is_member_pointer;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

23.15.4.3 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 \( \text{declval<T>()} \) for any type \( T \) is considered to be a trivial (6.7, Clause 15) function call that is not an odr-use (6.2) of \( \text{declval} \) in the context of the corresponding definition notwithstanding the restrictions of 23.2.6.

Table 42 — 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;</td>
<td>T is const-qualified</td>
<td>(6.7.3)</td>
</tr>
<tr>
<td>struct is_const;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is volatile-qualified</td>
<td></td>
</tr>
<tr>
<td>struct is_volatile;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is a trivial type</td>
<td></td>
</tr>
<tr>
<td>struct is_trivial;</td>
<td></td>
<td>remove_all_extents_(-t)&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is a trivially copyable type</td>
<td></td>
</tr>
<tr>
<td>struct is_trivially_copyable;</td>
<td></td>
<td>remove_all_extents_(-t)&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is a standard-layout type</td>
<td></td>
</tr>
<tr>
<td>struct is_standard_layout;</td>
<td></td>
<td>remove_all_extents_(-t)&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is a class type, but not a</td>
<td></td>
</tr>
<tr>
<td>struct is_empty;</td>
<td>union type, with no non-static data members other than bit-fields of length 0, no virtual member functions, no virtual base classes, and no base class ( B ) for which ( \text{is_empty_v}&lt;B&gt; ) is false.</td>
<td>If ( T ) is a non-union class type, ( T ) shall be a complete type.</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is a polymorphic class</td>
<td></td>
</tr>
<tr>
<td>struct is_polymorphic;</td>
<td></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;</td>
<td>T is an abstract class</td>
<td></td>
</tr>
<tr>
<td>struct is_abstract;</td>
<td></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;</td>
<td>T is a class type marked with the class-virt-specifier final (Clause 12). [ Note: A union is a class type that can be marked with final. — end note ]</td>
<td>If ( T ) is a class type, ( T ) shall be a complete type.</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>T is an aggregate type</td>
<td></td>
</tr>
<tr>
<td>struct is_aggregate;</td>
<td></td>
<td>remove_all_extents_(-t)&lt;T&gt; shall be a complete type or cv void.</td>
</tr>
</tbody>
</table>
Table 42 — 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_signed;</td>
<td>If <code>is_arithmetic_v&lt;T&gt;</code> is true, the same result as ( T(-1) &lt; T(0) ); otherwise, false</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_unsigned;</td>
<td>If <code>is_arithmetic_v&lt;T&gt;</code> is true, the same result as ( T(0) &lt; T(-1) ); otherwise, false</td>
<td></td>
</tr>
<tr>
<td>template&lt;class T, class... Args&gt; struct is_constructible;</td>
<td>For a function type ( T ) or for a <code>cv void</code> type ( T ), <code>is_constructible_v&lt;T, Args...&gt;</code> is false, otherwise see below</td>
<td><code>T</code> and all types in the parameter pack ( Args ) shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_default_constructible;</td>
<td><code>is_constructible_v&lt;T&gt;</code> is true.</td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_copy_constructible;</td>
<td>For a referenceable type ( T ) (20.3.18), the same result as <code>is_constructible_v&lt;T, const T&amp;&gt;</code>, otherwise false.</td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_move_constructible;</td>
<td>For a referenceable type ( T ), the same result as <code>is_constructible_v&lt;T, T&amp;&amp;&gt;</code>, otherwise false.</td>
<td><code>T</code> shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_assignable;</td>
<td>The expression <code>declval&lt;T&gt;() = declval&lt;U&gt;()</code> is well-formed when treated as an unevaluated operand (8.2). 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.</td>
<td><code>T</code> and ( U ) shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>

\[ \text{Note: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed.} \]
<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt; &lt;br&gt; struct is_copy_assignable;</td>
<td>For a referenceable type T, the same result as <code>isAssignable_v&lt;T&amp;, const T&amp;&gt;</code>, otherwise false.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; &lt;br&gt; struct is_move_assignable;</td>
<td>For a referenceable type T, the same result as <code>isAssignable_v&lt;T&amp;, T&amp;&amp;&gt;</code>, otherwise false.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; &lt;br&gt; struct is_swappable_with;</td>
<td>The expressions <code>swap(declval&lt;T&gt;(), declval&lt;U&gt;())</code> and <code>swap(declval&lt;U&gt;(), declval&lt;T&gt;())</code> are each well-formed when treated as an unevaluated operand (8.2) in an overload-resolution context for swappable values (20.5.3.2). Access checking is performed as if in a context unrelated to T and U. Only the validity of the immediate context of the swap expressions is considered. [Note: The compilation of the expressions can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note]</td>
<td>T and U shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; &lt;br&gt; struct is_swappable;</td>
<td>For a referenceable type T, the same result as <code>isSwappable_with_v&lt;T&amp;, T&amp;&gt;</code>, otherwise false.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; &lt;br&gt; struct is_destructible;</td>
<td>Either T is a reference type, or T is a complete object type for which the expression <code>declval&lt;U&amp;&gt;() .~U()</code> is well-formed when treated as an unevaluated operand (8.2), where U is <code>remove_all_extents_t&lt;T&gt;</code>.</td>
<td>T shall be a complete type, <code>cv void</code>, or an array of unknown bound.</td>
</tr>
</tbody>
</table>
Table 42 — Type property predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Preconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T, class... Args&gt; struct is_trivially_constructible;</td>
<td>is_constructible_v&lt;T, Args...&gt; is true and the variable definition for is_constructible, as defined below, is known to call no operation that is not trivial (6.7, Clause 15).</td>
<td>T and all types in the parameter pack Args shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_trivially_default_constructible;</td>
<td>is_trivially_constructible_v&lt;T&gt; is true.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<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, 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, Clause 15).</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 (8.5.2.7).</td>
<td>T and all types in the 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>
</tbody>
</table>
Table 42 — 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_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>
<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>isAssignable_v&lt;T, U&gt; is true and the assignment is known not to throw any exceptions (8.5.2.7).</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_copy_assignable;</td>
<td>For a referenceable type T, the same result as is_nothrow_assignable_v&lt;T&amp;, const T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_move_assignable;</td>
<td>For a referenceable type T, the same result as is_nothrow_assignable_v&lt;T&amp;, T&amp;&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T, class U&gt; struct is_nothrow_swappable_with;</td>
<td>is_swappable_with_v&lt;T, U&gt; is true and each swap expression of the definition of is_swappable_with_v&lt;T, U&gt; is known not to throw any exceptions (8.5.2.7).</td>
<td>T and U shall be complete types, cv void, or arrays of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_swappable;</td>
<td>For a referenceable type T, the same result as is_nothrow_swappable_v&lt;T&amp;, T&amp;&gt;, otherwise false.</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct is_nothrow_destructible;</td>
<td>is_destructible_v&lt;T&gt; is true and the indicated destructor is known not to throw any exceptions (8.5.2.7).</td>
<td>T shall be a complete type, cv void, or an array of unknown bound.</td>
</tr>
<tr>
<td>template&lt;class T&gt; struct has_virtualDestructor;</td>
<td>T has a virtual destructor (15.4)</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_t&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>

5 [Example:
is_const_v<const volatile int>     // true
is_const_v<const int*>            // false
is_const_v<const int&>            // false
is_const_v<int[3]>                // false
is_const_v<const int[3]>          // true

— end example]

6 [Example:
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]

7 [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]

8 The predicate condition for a template specialization is_constructible<T, Args...> shall be satisfied if and only if the following variable definition would be well-formed for some invented variable $t$:

   T t(declval<Args>()...);

[Note: These tokens are never interpreted as a function declaration. —end note] Access checking is performed as if in a context unrelated to $T$ and any of the Args. Only the validity of the immediate context of the variable initialization is considered. [Note: The evaluation of the initialization can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. —end note]

9 The predicate condition for a template specialization has_unique_object_representations<T> shall be satisfied if and only if:

   (9.1) $T$ is trivially copyable, and
   (9.2) any two objects of type $T$ with the same value have the same object representation, where two objects of array or non-union class type are considered to have the same value if their respective sequences of direct subobjects have the same values, and two objects of union type are considered to have the same value if they have the same active member and the corresponding members have the same value.

The set of scalar types for which this condition holds is implementation-defined. [Note: If a type has padding bits, the condition does not hold; otherwise, the condition holds true for unsigned integral types. —end note]

### 23.15.5 Type property queries

This subclause contains templates that may be used to query properties of types at compile time.

<table>
<thead>
<tr>
<th>Template</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;class T&gt;</td>
<td>alignof(T).</td>
</tr>
<tr>
<td>struct alignment_of;</td>
<td>Requires: alignof(T) shall be a valid expression (8.5.2.6)</td>
</tr>
<tr>
<td>template&lt;class T&gt;</td>
<td>If $T$ names an array type, an integer value representing the number of dimensions of $T$; otherwise, 0.</td>
</tr>
<tr>
<td>struct rank;</td>
<td></td>
</tr>
</tbody>
</table>

§ 23.15.5 628
### Table 43 — Type property queries (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{template&lt;class T, unsigned I = 0&gt; struct extent;}</td>
<td>If ( T ) is not an array type, or if it has rank less than or equal to ( I ), or if ( I ) is 0 and ( T ) has type &quot;array of unknown bound of ( U )”, then 0; otherwise, the bound (11.3.4) of the ( I )’th dimension of ( T ), where indexing of ( I ) is zero-based.</td>
</tr>
</tbody>
</table>

2 Each of these templates shall be a UnaryTypeTrait (23.15.1) with a base characteristic of \texttt{integral_constant<size_t, Value>}.  

3 \[\text{Example:}\]

\[
// the following assertions hold:  
assert(rank_v<int> == 0);  
assert(rank_v<int[2]> == 1);  
assert(rank_v<int[][4]> == 2);  
\]

\[-end example\]

4 \[\text{Example:}\]

\[
// the following assertions hold:  
assert(extent_v<int> == 0);  
assert(extent_v<int[2]> == 2);  
assert(extent_v<int[2][4]> == 2);  
assert(extent_v<int[][4]> == 0);  
assert((extent_v<int, 1>) == 0);  
assert((extent_v<int[2], 1>) == 0);  
assert((extent_v<int[2][4], 1>) == 4);  
assert((extent_v<int[][4], 1>) == 4);  
\]

\[-end example\]

### 23.15.6 Relationships between types \[\text{meta.rel}\]

1 This subclause contains templates that may be used to query relationships between types at compile time.

2 Each of these templates shall be a BinaryTypeTrait (23.15.1) with a base characteristic of \texttt{true_type} if the corresponding condition is true, otherwise \texttt{false_type}.  

### Table 44 — Type relationship predicates

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{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>\texttt{template&lt;class Base, class Derived&gt; struct is_base_of;}</td>
<td>Base is a base class of \texttt{Derived} (Clause 13) without regard to cv-qualifiers or \texttt{Base} and \texttt{Derived} are not unions and name the same class type without regard to cv-qualifiers</td>
<td>If \texttt{Base} and \texttt{Derived} are non-union class types and are not possibly cv-qualified versions of the same type, \texttt{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>\texttt{template&lt;class From, class To&gt; struct is_convertible;}</td>
<td>see below</td>
<td>From and \texttt{To} shall be complete types, arrays of unknown bound, or \texttt{cv void} types.</td>
</tr>
<tr>
<td>\texttt{template&lt;class Fn, class... ArgTypes&gt; struct is_invocable;}</td>
<td>The expression \texttt{\texttt{INVOLVE}(declval&lt;\texttt{Fn}&gt;(), declval&lt;\texttt{ArgTypes}&gt;(),...)} is well-formed when treated as an unevaluated operand</td>
<td>\texttt{Fn} and all types in the parameter pack \texttt{ArgTypes} shall be complete types, \texttt{cv void}, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>

§ 23.15.6 629
### Table 44 — Type relationship predicates (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Condition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class R, class Fn, class... ArgTypes&gt;</code> <code>struct is_invocable_r;</code></td>
<td>The expression <code>INVOKE&lt;R&gt;(declval&lt;Fn&gt;(), declval&lt;ArgTypes&gt;()...)</code> is well-formed when treated as an unevaluated operand</td>
<td><code>Fn</code>, <code>R</code>, and all types in the parameter pack <code>ArgTypes</code> shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class Fn, class... ArgTypes&gt;</code> <code>struct is_nothrow_invocable;</code></td>
<td><code>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</code></td>
<td><code>Fn</code> and all types in the parameter pack <code>ArgTypes</code> shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
<tr>
<td><code>template&lt;class R, class Fn, class... ArgTypes&gt;</code> <code>struct is_nothrow_invocable_r;</code></td>
<td><code>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</code></td>
<td><code>Fn</code>, <code>R</code>, and all types in the parameter pack <code>ArgTypes</code> shall be complete types, <code>cv void</code>, or arrays of unknown bound.</td>
</tr>
</tbody>
</table>

3 For the purpose of defining the templates in this subclause, a function call expression `declval<T>()` for any type `T` is considered to be a trivial (6.7, Clause 15) function call that is not an odr-use (6.2) of `declval` in the context of the corresponding definition notwithstanding the restrictions of 23.2.6.

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, const B> // true
is_base_of_v<D, 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 (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]
23.15.7 Transformations between types

This subclause contains templates that may be used to transform one type to another following some predefined rule.

Each of the templates in this subclause shall be a TransformationTrait (23.15.1).

23.15.7.1 Const-volatile modifications

Table 45 — 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;const 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 const 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>

23.15.7.2 Reference modifications

Table 46 — 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 (20.3.18) then the member typedef type names T &amp;; otherwise, type names T. [Note: This rule reflects the semantics of reference collapsing (11.3.2). — 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 (11.3.2). 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>

23.15.7.3 Sign modifications

§ 23.15.7.3
Table 47 — Sign modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T&gt; struct make_signed;</code></td>
<td>If <code>T</code> names a (possibly cv-qualified) signed integer type (6.7.1) then the member typedef <code>type</code> names the type <code>T</code>; otherwise, if <code>T</code> names a (possibly cv-qualified) unsigned integer type then <code>type</code> names the corresponding signed integer type, with the same cv-qualifiers as <code>T</code>; otherwise, <code>type</code> names the signed integer type with smallest rank (6.7.4) for which <code>sizeof(T) == sizeof(type)</code>, with the same cv-qualifiers as <code>T</code>. <strong>Requires:</strong> <code>T</code> shall be a (possibly cv-qualified) integral type or enumeration but not a <code>bool</code> type.</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct make_unsigned;</code></td>
<td>If <code>T</code> names a (possibly cv-qualified) unsigned integer type (6.7.1) then the member typedef <code>type</code> names the type <code>T</code>; otherwise, if <code>T</code> names a (possibly cv-qualified) signed integer type then <code>type</code> names the corresponding unsigned integer type, with the same cv-qualifiers as <code>T</code>; otherwise, <code>type</code> names the unsigned integer type with smallest rank (6.7.4) for which <code>sizeof(T) == sizeof(type)</code>, with the same cv-qualifiers as <code>T</code>. <strong>Requires:</strong> <code>T</code> shall be a (possibly cv-qualified) integral type or enumeration but not a <code>bool</code> type.</td>
</tr>
</tbody>
</table>

23.15.7.4 Array modifications

Table 48 — Array modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T&gt; struct remove_extent;</code></td>
<td>If <code>T</code> names a type “array of <code>U</code>”, the member typedef <code>type</code> shall be <code>U</code>, otherwise <code>T</code>. [Note: For multidimensional arrays, only the first array dimension is removed. For a type “array of const <code>U</code>”, the resulting type is <code>const U</code>. — end note]</td>
</tr>
<tr>
<td><code>template&lt;class T&gt; struct remove_all_extents;</code></td>
<td>If <code>T</code> is “multi-dimensional array of <code>U</code>”, the resulting member typedef <code>type</code> is <code>U</code>, otherwise <code>T</code>.</td>
</tr>
</tbody>
</table>

1 **Example:**

```c
// the following assertions hold:
assert((is_same_v<remove_extent_t<int>, int>));
assert((is_same_v<remove_extent_t<int[2]>, int>));
assert((is_same_v<remove_extent_t<int[2][3]>, int[3]>));
assert((is_same_v<remove_extent_t<int[1][3]>, int[3]>));
```

—end example

2 **Example:**

```c
// the following assertions hold:
assert((is_same_v<remove_all_extents_t<int>, int>));
assert((is_same_v<remove_all_extents_t<int[2]>, int>));
assert((is_same_v<remove_all_extents_t<int[2][3]>, int[]>));
assert((is_same_v<remove_all_extents_t<int[1][3]>, int[]>));
```

—end example

23.15.7.5 Pointer modifications

Table 49 — Pointer modifications

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;class T&gt; struct remove_pointer;</code></td>
<td>If <code>T</code> has type “(possibly cv-qualified) pointer to <code>T1</code>” then the member typedef <code>type</code> names <code>T1</code>; otherwise, it names <code>T</code>.</td>
</tr>
</tbody>
</table>
### Template Comments

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| `template<class T>
struct add_pointer;` | If `T` names a referenceable type (20.3.18) or a `cv void` type then the member typedef `type` names the same type as `remove_reference_t<T>*`; otherwise, `type` names `T`. |

#### Other transformations

**Table 50 — Other transformations**

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| `template<size_t Len,
size_t Align = default-alignment>
struct aligned_storage;` | The value of `default-alignment` shall be the most stringent alignment requirement for any C++ object type whose size is no greater than `Len (6.7)`. The member typedef `type` shall be a trivial 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`. |
| `template<size_t Len,
class... Types>
struct aligned_union;` | The member typedef `type` shall be a trivial 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 parameter pack `Types` shall be a complete object type. |
| `template<class T>
struct remove_cvref;` | The member typedef `type` names the same type as `remove_cv_t<remove_reference_t<T>>`. |
| `template<class T>
struct decay;` | Let `U` be `remove_reference_t<T>`. If `is_array_v<U>` is true, the member typedef `type` shall equal `remove_extent_t<U>`. If `is_function_v<U>` is true, the member typedef `type` shall equal `add_pointer_t<U>`. Otherwise the member typedef `type` equals `remove_cv_t<U>`.

[Note: This behavior is similar to the lvalue-to-rvalue (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) conversions applied when an lvalue expression is used as an rvalue, but also strips `cv`-qualifiers from class types in order to more closely model by-value argument passing. — end note] |
| `template<bool B, class T = void>
struct enable_if;` | If `B` is true, the member typedef `type` shall equal `T`; otherwise, there shall be no member `type`. |
| `template<bool B, class T, class F>
struct conditional;` | If `B` is true, the member typedef `type` shall equal `T`. If `B` is false, the member typedef `type` shall equal `F`. |
| `template<class... T>
struct common_type;` | Unless this trait is specialized (as specified in Note B, below), the member `type` 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 parameter pack `T` shall be complete, `cv` `void`, or an array of unknown bound. |
| `template<class T>
struct underlying_type;` | The member typedef `type` names the underlying type of `T`. **Requires:** `T` shall be a complete enumeration type (10.2) |
Table 50 — Other transformations (continued)

<table>
<thead>
<tr>
<th>Template</th>
<th>Comments</th>
</tr>
</thead>
</table>
| `template<class Fn, class... ArgTypes>
 struct invoke_result;` | If the expression `INVOKE(declval<Fn>(), decval<ArgTypes>()...)` is well-formed when treated as an unevaluated operand (8.2), the member typedef `type` names the type `decltype(INVOKE(declval<Fn>(), declval<ArgTypes>()...));` otherwise, there shall be no member `type`. Access checking is performed as if in a context unrelated to `Fn` and `ArgTypes`. Only the validity of the immediate context of the expression is considered. 
[Note: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the “immediate context” and can result in the program being ill-formed. — end note]

Requires: `Fn` and all types in the 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]
```

It is implementation-defined whether any extended alignment is supported (6.6.5).

3 Note A: For the `common_type` trait applied to a parameter pack `T` of types, the member `type` shall be either defined or not present as follows:

(3.1) — If `sizeof...(T)` is zero, there shall be no member `type`.
(3.2) — 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`.
(3.3) — If `sizeof...(T)` is two, let the first and second types constituting `T` be denoted by `T1` and `T2`, respectively, and let `D1` and `D2` denote the same types as `decay_t<T1>` and `decay_t<T2>`, respectively.
(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) — Otherwise, let `C` denote the same type, if any, as `decay_t<decltype(false ? declval<D1>() : declval<D2>())>`.

[Note: This will not apply if there is a specialization `common_type<D1, D2>`. — end note]

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...>`. Otherwise, there shall be no member `type`.

4 Note B: Notwithstanding the provisions of 23.15.2, and pursuant to 20.5.4.2.1, 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 C 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 common_type_t<T2, T1>. No diagnostic is required for a violation of this Note’s rules.

[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&>);
static_assert(is_same_v<invoke_result_t<PMD, const S*>, const char&>);
```

—end example]

23.15.8 Logical operator traits

This subclause describes type traits for applying logical operators to other type traits.

```
template<class... B> struct conjunction : see below { };
```

The class template conjunction forms the logical conjunction of its template type arguments.

```
For a specialization conjunction<B1, ..., BN>, if there is a template type argument Bi for which bool(Bi::value) is false, then instantiating conjunction<B1, ..., BN>::value does not require the instantiation of Bj::value for j > i. [Note: This is analogous to the short-circuiting behavior of the built-in operator &&. —end note]
```

Every template type argument for which Bi::value is instantiated shall be usable as a base class and shall have a member value which is convertible to bool, is not hidden, and is unambiguously available in the type.

The specialization conjunction<B1, ..., BN> has a public and unambiguous base that is either

- the first type Bi in the list true_type, B1, ..., BN for which bool(Bi::value) is false, or
- if there is no such Bi, the last type in the list.

[Note: This means a specialization of conjunction does not necessarily inherit from either true_type or false_type. —end note]

The member names of the base class, other than conjunction and operator=, shall not be hidden and shall be unambiguously available in conjunction.

```
template<class... B> struct disjunction : see below { };
```

The class template disjunction forms the logical disjunction of its template type arguments.

```
For a specialization disjunction<B1, ..., BN>, if there is a template type argument Bi for which bool(Bi::value) is true, then instantiating disjunction<B1, ..., BN>::value does not require the instantiation of Bj::value for j > i. [Note: This is analogous to the short-circuiting behavior of the built-in operator ||. —end note]
```

Every template type argument for which Bi::value is instantiated shall be usable as a base class and shall have a member value which is convertible to bool, is not hidden, and is unambiguously available in the type.

The specialization disjunction<B1, ..., BN> has a public and unambiguously available in the type.
(10.1) 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
(10.2) if there is no such $B_i$, the last type in the list.

[Note: This means a specialization of $\text{disjunction}$ does not necessarily inherit from either $\text{true_type}$ or $\text{false_type}$. — end note]

11 The member names of the base class, other than $\text{disjunction}$ and $\text{operator=}$, shall not be hidden and shall be unambiguously available in $\text{disjunction}$.

template<class B> struct negation : see below { };

12 The class template $\text{negation}$ forms the logical negation of its template type argument. The type $\text{negation<B>}$ is a $\text{UnaryTypeTrait}$ with a base characteristic of $\text{bool_constant<!bool(B::value)>}$.

23.15.9 Endian [meta.endian]

1 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.

```cpp
enum class endian {
    little = see below,
    big = see below,
    native = see below
};
```

2 If all scalar types have size 1 byte, then all of $\text{endian::little}$, $\text{endian::big}$, and $\text{endian::native}$ have the same value. Otherwise, $\text{endian::little}$ is not equal to $\text{endian::big}$. If all scalar types are big-endian, $\text{endian::native}$ is equal to $\text{endian::big}$. If all scalar types are little-endian, $\text{endian::native}$ is equal to $\text{endian::little}$. Otherwise, $\text{endian::native}$ is not equal to either $\text{endian::big}$ or $\text{endian::little}$.

23.16 Compile-time rational arithmetic [ratio]

23.16.1 In general [ratio.general]

1 This subclause describes the ratio library. It provides a class template $\text{ratio}$ which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type $\text{intmax_t}$.

2 Throughout this subclause, the names of template parameters are used to express type requirements. If a template parameter is named $R_1$ or $R_2$, and the template argument is not a specialization of the $\text{ratio}$ template, the program is ill-formed.

23.16.2 Header $<\text{ratio}>$ synopsis [ratio.syn]

```cpp
namespace std {
    // 23.16.3, class template ratio
    template<intmax_t N, intmax_t D = 1> class ratio;

    // 23.16.4, 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;

    // 23.16.5, 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;
}
```

§ 23.16.2 636
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;

// 23.16.6, convenience SI typedefs
using yocto = ratio<1, 1'000'000'000'000'000'000'000'000';> // see below
using zepto = ratio<1, 1'000'000'000'000'000'000'000';> // see below
using atto = ratio<1, 1'000'000'000'000'000';>;
using femto = ratio<1, 1'000'000';>;
using pico = ratio<1, 1'000';>;
using nano = ratio<1, 1';>;
using micro = ratio<1, 1';>;
using milli = ratio<1, 1';>;
using centi = ratio<1, 10';>;
using deci = ratio<1, 10';>;
using deca = ratio<1, 10';>;
using hecto = ratio<1, 100';>;
using kilo = ratio<1, 1'000';>;
using mega = ratio<1', 1';>;
using giga = ratio<1', 1';>;
using tera = ratio<1', 1';>;
using peta = ratio<1', 1';>;
using exa = ratio<1', 1'>; // see below
using yotta = ratio<1', 1'>; // see below

23.16.3 Class template ratio

namespace std {
    template<intmax_t N, intmax_t D = 1> class ratio {
    public:
        static constexpr intmax_t num;
        static constexpr intmax_t den;
        using type = ratio<num, den>;
    }
}

1 If the template argument D is zero or the absolute values of either of the template arguments N and D is not representable by type intmax_t, the program is ill-formed. [Note: These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute value which is positive. In a two's complement representation, this excludes the most negative value. —end note]

2 The static data members num and den shall have the following values, where gcd represents the greatest common divisor of the absolute values of N and D:

(2.1) — num shall have the value sign(N) * sign(D) * abs(N) / gcd.
(2.2) — den shall have the value abs(D) / gcd.

23.16.4 Arithmetic on ratios

1 Each of the alias templates ratio_add, ratio_subtract, ratio_multiply, and ratio_divide denotes the result of an arithmetic computation on two ratios R1 and R2. With X and Y computed (in the absence of arithmetic overflow) as specified by Table 51, each alias denotes a ratio<X, Y> 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 \( \text{intmax}_t \), 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 \( \text{intmax}_t \), the program is ill-formed unless the implementation yields correct values of \( U \) and \( V \).

### Table 51 — Expressions used to perform ratio arithmetic

<table>
<thead>
<tr>
<th>Type</th>
<th>Value of ( X )</th>
<th>Value of ( Y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio_add&lt;( R_1, R_2 )&gt;</td>
<td>( R_1::num \times R_2::den + R_1::den \times R_2::den )</td>
<td>( R_1::den \times R_2::num )</td>
</tr>
<tr>
<td>ratio_subtract&lt;( R_1, R_2 )&gt;</td>
<td>( R_1::num \times R_2::den - R_1::den \times R_2::den )</td>
<td>( R_2::num \times R_1::den )</td>
</tr>
<tr>
<td>ratio_multiply&lt;( R_1, R_2 )&gt;</td>
<td>( R_1::num \times R_2::num )</td>
<td>( R_1::den \times R_2::den )</td>
</tr>
<tr>
<td>ratio_divide&lt;( R_1, R_2 )&gt;</td>
<td>( R_1::num \times R_2::den )</td>
<td>( R_1::num \times R_2::den )</td>
</tr>
</tbody>
</table>

[Example:

```cpp
class ratio {
    int num, den;
    ratio(int n, int d) : num(n), den(d) {}
    ratio& operator+=(const ratio& other) {
        num += other.num * den;
        den += other.den * num;
        return *this;
    }
    ratio& operator-=(const ratio& other) {
        num -= other.num * den;
        den -= other.den * num;
        return *this;
    }
    ratio& operator*=(const ratio& other) {
        num *= other.num;
        den *= other.den;
        return *this;
    }
    ratio& operator/=(const ratio& other) {
        den /= other.num;
        num /= other.den;
        return *this;
    }
};

ratio a(1, 3), b(1, 6);
ratio c = a + b; // Should be equal to (1 + 1)/2
ratio d = a - b; // Should be equal to (1 - 1)/2
ratio e = a * b; // Should be equal to (1 * 1)/2
ratio f = a / b; // Should be equal to (1 * 6)/(1 * 3)
```

—end example]

23.16.5 Comparison of ratios

```cpp
template<class R1, class R2>
    struct ratio_equal : bool_constant<R1::num == R2::num && R1::den == R2::den> { };
template<class R1, class R2>
    struct ratio_not_equal : bool_constant<!ratio_equal_v<R1, R2>> { };
template<class R1, class R2>
    struct ratio_less : bool_constant<see below> { };
```

If \( R_1::num \times R_2::den \) is less than \( R_2::num \times R_1::den \), ratio_less<\( R_1, R_2 \)> 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.

```cpp
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>> { };
```

23.16.6 SI types for ratio

1 For each of the typedef-names yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by \( \text{intmax}_t \), the typedef shall be defined; if either of the constants is not representable by \( \text{intmax}_t \), the typedef shall not be defined.
23.17 Time utilities

23.17.1 In general

This subclause describes the chrono library (23.17.2) and various C functions (23.17.8) that provide generally useful time utilities.

23.17.2 Header <chrono> synopsis

namespace std {
    namespace chrono {
        // 23.17.5, class template duration
        template<class Rep, class Period = ratio<1>> class duration;

        // 23.17.6, class template time_point
        template<class Clock, class Duration = typename Clock::duration> class time_point;
    }

    // 23.17.4.3, 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 {
        // 23.17.4, 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;

        // 23.17.5.5, 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);

        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 Rep2, class Period>
        constexpr duration<common_type_t<Rep1, Rep2>, Period>
            operator/(const duration<Rep1, Period>& d, const Rep2& s);

        template<class Rep1, class Period1, class Rep2, class Period2>
        constexpr common_type_t<duration<Rep1, Period1>, duration<Rep2, Period2>>
            operator%(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

        // 23.17.5.6, duration comparisons
        template<class Rep1, class Period1, class Rep2, class Period2>
        constexpr bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
    }

§ 23.17.2
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);

// 23.17.5.7, duration_cast
template<class ToDuration, class Rep, class Period>
constexpr ToDuration duration_cast(const duration<Rep, Period>& d);

// 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>;

// 23.17.6.5, 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 Clock, class Duration1, class Rep2, class Period2>
constexpr time_point<Clock, common_type_t<duration<Rep2, Period2>, Duration1>>
operator+(const duration<Rep2, Period2>& lhs, const time_point<Clock, Duration1>& 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);

// 23.17.6.6, 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);
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);

// 23.17.6.7, time_point_cast
template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration>
time_point_cast(const time_point<Clock, Duration>& t);

// 23.17.5.9, specialized algorithms
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);

// 23.17.5.8, suffixes for duration literals
constexpr chrono::hours operator"h(unsigned long long);
constexpr chrono::duration<unsigned long long, ratio<3600, 1>> operator"h(long double);
constexpr chrono::minutes operator"min(unsigned long long);
constexpr chrono::duration<unsigned long long, ratio<60, 1>> operator"min(long double);
constexpr chrono::seconds operator"s(unsigned long long);
constexpr chrono::duration<unsigned long long, ratio<1, 1>> operator"s(long double);
constexpr chrono::milliseconds operator"ms(unsigned long long);
constexpr chrono::duration<unsigned long long, milli> operator"ms(long double);
constexpr chrono::microseconds operator"us(unsigned long long);
constexpr chrono::duration<unsigned long long, micro> operator"us(long double);
constexpr chrono::nanoseconds operator"ns(unsigned long long);
constexpr chrono::duration<unsigned long long, nano> operator"ns(long double);

namespace chrono {
using namespace literals::chrono_literals;
}

namespace chrono {

// 23.17.3  Clock requirements

A clock is a bundle consisting of a duration, a time_point, and a function now() to get the current time_point. The origin of the clock's time_point is referred to as the clock's epoch. A clock shall meet the requirements in Table 52.

In Table 52 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 52 — Clock requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1::rep</td>
<td>An arithmetic type or a class emulating an arithmetic type</td>
<td>The representation type of C1::duration.</td>
</tr>
<tr>
<td>C1::period</td>
<td>a specialization of ratio</td>
<td>The tick period of the clock in seconds.</td>
</tr>
<tr>
<td>C1::duration</td>
<td>chrono::duration&lt;C1::rep, C1::period&gt;</td>
<td>The duration type of the clock.</td>
</tr>
<tr>
<td>C1::time_point</td>
<td>chrono::time_point&lt;C1&gt; or chrono::time_point&lt;C2, C1::duration&gt;</td>
<td>The time_point type of the clock. C1 and C2 shall refer to the same epoch.</td>
</tr>
<tr>
<td>C1::is_steady</td>
<td>const bool</td>
<td>true if t1 &lt;= t2 is always true and the time between clock ticks is constant, otherwise false.</td>
</tr>
<tr>
<td>C1::now()</td>
<td>C1::time_point</td>
<td>Returns a time_point object representing the current point in time.</td>
</tr>
</tbody>
</table>

3 [Note: 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 TrivialClock requirements if:

(4.1) — TC satisfies the Clock requirements (23.17.3),
(4.2) — the types TC::rep, TC::duration, and TC::time_point satisfy the requirements of EqualityComparable (Table 20), LessThanComparable (Table 21), DefaultConstructible (Table 22), CopyConstructible (Table 24), CopyAssignable (Table 26), Destructible (Table 27), and the requirements of numeric types (29.3). [Note: This means, in particular, that operations on these types will not throw exceptions. —end note]
(4.3) — values of the types TC::rep, TC::duration, and TC::time_point are swappable (20.5.3.2),
(4.4) — the function TC::now() does not throw exceptions, and
(4.5) — the type TC::time_point::clock meets the TrivialClock requirements, recursively.

23.17.4 Time-related traits

23.17.4.1 treat_as_floating_point

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]

23.17.4.2 duration_values

template<class Rep>
struct duration_values {
    static constexpr Rep zero();
    static constexpr Rep min();
    static constexpr Rep max();
};  
1 The duration template uses the duration_values trait to construct special values of the durations 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();
```

Returns: `Rep(0)`. [Note: `Rep(0)` is specified instead of `Rep()` because `Rep()` may have some other meaning, such as an uninitialized value. —end note]

Remarks: The value returned shall be the additive identity.

```cpp
static constexpr Rep min();
```

Returns: `numeric_limits<Rep>::lowest()`.

Remarks: The value returned shall compare less than or equal to `zero()`.

```cpp
static constexpr Rep max();
```

Returns: `numeric_limits<Rep>::max()`.

Remarks: The value returned shall compare greater than `zero()`.

### 23.17.4.3 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*.

### 23.17.5 Class template `duration`

A *duration* type measures time between two points in time (*time_points*). A *duration* has a representation which holds a count of ticks and a tick period. The tick period is the amount of time which occurs from one tick to the next, in units of seconds. It is expressed as a rational constant using the template `ratio`.

```cpp
namespace std::chrono {
    template<class Rep, class Period = ratio<1>>
    class duration {
        public:
            using rep = Rep;
            using period = typename Period::type;

        private:
            rep rep_; // exposition only

        public:
            // 23.17.5.1, 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);
```

§ 23.17.5 643
~duration() = default;
duration(const duration&) = default;
duration& operator=(const duration&) = default;

// 23.17.5.2, observer
cnstexpr rep count() const;

// 23.17.5.3, arithmetic
cnstexpr common_type_t<duration> operator+() const;
cnstexpr common_type_t<duration> operator-() const;
cnstexpr duration& operator++();
cnstexpr duration operator++(int);
cnstexpr duration& operator--();
cnstexpr duration operator--(int);

cnstexpr duration& operator+=(const duration& d);
cnstexpr duration& operator-=(const duration& d);

cnstexpr duration& operator*=(const rep& rhs);
cnstexpr duration& operator/=(const rep& rhs);
cnstexpr duration& operator%=(const rep& rhs);
cnstexpr duration& operator%=(const duration& rhs);

// 23.17.5.4, special values
static constexpr duration zero();
static constexpr duration min();
static constexpr duration max();

};

2 Rep shall be an arithmetic type or a class emulating an arithmetic type. If duration is instantiated with a duration type as the argument for the template parameter Rep, the program is ill-formed.

3 If Period is not a specialization of ratio, the program is ill-formed. If Period::num is not positive, the program is ill-formed.

4 Members of duration shall not throw exceptions other than those thrown by the indicated operations on their representations.

5 The defaulted copy constructor of duration shall be a constexpr function if and only if the required initialization of the member rep_ for copy and move, respectively, would satisfy the requirements for a constexpr function.

6 [Example:
   duration<long, ratio<60>> d0;  // holds a count of minutes using a long
   duration<long long, milli> d1;  // holds a count of milliseconds using a long long
   duration<double, ratio<1, 30>> d2; // holds a count with a tick period of \frac{1}{30} of a second
   // (30 Hz) using a double
   — end example]

23.17.5.1 duration constructors

template<class Rep2>
cnstexpr explicit duration(const Rep2& r);

1 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.

[Example:
   duration<int, milli> d(3);  // OK
   duration<int, milli> d(3.5);  // error
   — end example]

2 Effects: Constructs an object of type duration.
Postconditions: 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<Rep2> 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().

23.17.5.2 duration observer

constexpr rep count() const;
Returns: rep_.

23.17.5.3 duration arithmetic

constexpr common_type_t<duration> operator+(duration) const;
Returns: common_type_t<duration>(*this).

constexpr common_type_t<duration> operator-(duration) 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.

constexpr duration& operator*=(const rep& rhs);
Effects: As if by: rep_ *= rhs;
Returns: *this.

§ 23.17.5.3
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.

23.17.5.4 duration special values

static constexpr duration zero();
Returns: duration(duration_values<rep>::zero()).

static constexpr duration min();
Returns: duration(duration_values<rep>::min()).

static constexpr duration max();
Returns: duration(duration_values<rep>::max()).

23.17.5.5 duration non-member arithmetic

In the function descriptions that follow, CD represents the return type of the function. CR(A, B) represents common_type_t<A, B>.

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 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 CR(Rep1, Rep2).
Returns: CD(CD(d).count() * s).

template<class Rep1, class Rep2, class Period>
constexpr duration<common_type_t<Rep1, Rep2>, Period>
operator*(const Rep1& s, const duration<Rep2, Period>& d);

Remarks: This operator shall not participate in overload resolution unless Rep1 is implicitly convertible to CR(Rep1, Rep2).
Returns: d * s.

template<class Rep1, class Rep2, class Period>
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 CR(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);

Returns: CD(lhs).count() / CD(rhs).count().

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr duration<common_type_t<Rep1, Rep2>, Period1>
operator%(const duration<Rep1, Period1>& d, const Rep2& s);

Remarks: This operator shall not participate in overload resolution unless Rep2 is implicitly convertible to CR(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()).

23.17.5.6 duration comparisons

1 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: CT(lhs).count() < CT(rhs).count().

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: rhs < lhs.

template<class Rep1, class Period1, class Rep2, class Period2>
constexpr bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

Returns: !(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).

23.17.5.7 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.
If \( CF::num == 1 \) and \( CF::den == 1 \), returns
\[
\text{ToDuration(static\_cast<typename ToDuration::rep>(d.count()))}
\]
otherwise, if \( CF::num != 1 \) and \( CF::den == 1 \), returns
\[
\text{ToDuration(static\_cast<typename ToDuration::rep>(}
\text{ \text{static\_cast<CR>(d.count())} \* \text{static\_cast<CR>(CF::num)}))}
\]
otherwise, if \( CF::num == 1 \) and \( CF::den != 1 \), returns
\[
\text{ToDuration(static\_cast<typename ToDuration::rep>(}
\text{ \text{static\_cast<CR>(d.count())} / \text{static\_cast<CR>(CF::den)}))}
\]
otherwise, returns
\[
\text{ToDuration(static\_cast<typename ToDuration::rep>(}
\text{ \text{static\_cast<CR>(d.count())} * \text{static\_cast<CR>(CF::num)} / \text{static\_cast<CR>(CF::den)})}
\]

[Note: This function does not use any implicit conversions; all conversions are done with \text{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]

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration floor(const duration<Rep, Period>& d);
```

Remarks: This function shall not participate in overload resolution unless \text{ToDuration} is a specialization of \text{duration}.

Returns: The greatest result \( t \) representable in \text{ToDuration} for which \( t <= d \).

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration ceil(const duration<Rep, Period>& d);
```

Remarks: This function shall not participate in overload resolution unless \text{ToDuration} is a specialization of \text{duration}.

Returns: The least result \( t \) representable in \text{ToDuration} for which \( t >= d \).

```cpp
template<class ToDuration, class Rep, class Period>
constexpr ToDuration round(const duration<Rep, Period>& d);
```

Remarks: This function shall not participate in overload resolution unless \text{ToDuration} is a specialization of \text{duration}, and \text{treat\_as\_floating\_point\_v<typename ToDuration::rep>} is \text{false}.

Returns: The value of \text{ToDuration} that is closest to \( d \). If there are two closest values, then return the value \( t \) for which \( t \% 2 == 0 \).

### 23.17.5.8 Suffixes for duration literals

This subclause describes literal suffixes for constructing duration literals. The suffixes \text{h}, \text{min}, \text{s}, \text{ms}, \text{us}, \text{ns} denote duration values of the corresponding types \text{hours}, \text{minutes}, \text{seconds}, \text{milliseconds}, \text{microseconds}, and \text{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 \text{chrono::duration} literal with an unspecified floating-point representation.

If any of these suffixes are applied to an integer literal and the resulting \text{chrono::duration} value cannot be represented in the result type because of overflow, the program is ill-formed.

[Example: The following code shows some duration literals.

```cpp
using namespace std::chrono_literals;
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;
```

—end example]

```cpp
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.

23.17.5.9 duration algorithms

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.

23.17.6 Class template time_point

namespace std::chrono {

template<class Clock, class Duration = typename Clock::duration>
class time_point {

public:
  using clock = Clock;
  using duration = Duration;
  using rep = typename duration::rep;
  using period = typename duration::period;

private:
  duration d_; // exposition only

public:
  // 23.17.6.1, 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);

  // 23.17.6.2, observer
  constexpr duration time_since_epoch() const;

  // 23.17.6.3, arithmetic
  constexpr time_point& operator+=(const duration& d);
  constexpr time_point& operator+=(const duration& d);

  // 23.17.6.4, special values
  static constexpr time_point min();
  static constexpr time_point max();

} // namespace std::chrono
1 Clock shall meet the Clock requirements (23.17.3).
2 If Duration is not an instance of duration, the program is ill-formed.

23.17.6.1 time_point constructors

csstexpr time_point();
Effects: Constructs an object of type time_point, initializing d_ with duration::zero(). Such a
time_point object represents the epoch.

csstexpr explicit time_point(const duration& d);
Effects: Constructs an object of type time_point, initializing d_ with d. Such a time_point object
represents the epoch + d.

template<class Duration2>
csstexpr time_point(const time_point<clock, Duration2>& t);
Remarks: This constructor shall not participate in overload resolution unless
Duration2 is implicitly
convertible to duration.
Effects: Constructs an object of type time_point, initializing d_ with t.time_since_epoch().

23.17.6.2 time_point observer

csstexpr duration time_since_epoch() const;
Returns: d_.

23.17.6.3 time_point arithmetic

csstexpr time_point& operator+=(const duration& d);
Effects: As if by: d_ += d;
Returns: *this.

csstexpr time_point& operator-=(const duration& d);
Effects: As if by: d_ -= d;
Returns: *this.

23.17.6.4 time_point special values

static csstexpr time_point min();
Returns: time_point(duration::min()).
static csstexpr time_point max();
Returns: time_point(duration::max()).

23.17.6.5 time_point non-member arithmetic

template<class Clock, class Duration1, class Rep2, class Period2>
csstexpr 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>
csstexpr 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>
csstexpr 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().

23.17.6.6 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);

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: !(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).

23.17.6.7 time_point_cast

template<class ToDuration, class Clock, class Duration>
constexpr time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

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())).

23.17.7 Clocks

The types defined in this subclause shall satisfy the TrivialClock requirements (23.17.3).

23.17.7.1 Class system_clock

Objects of class system_clock represent wall clock time from the system-wide realtime clock.

namespace std::chrono {
    class system_clock {
        public:
            using rep = unspecified;
            using period = ratio<unspecified, unspecified>;
            using duration = chrono::duration<rep, period>;
            using time_point = chrono::time_point<system_clock>;
            static constexpr bool is_steady = unspecified;

            static time_point now() noexcept;
            static time_t to_time_t(const time_point& t) noexcept;
            static time_point from_time_t(time_t t) noexcept;
    };
}

using system_clock::rep = unspecified;

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;

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;

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.

23.17.7.2 Class steady_clock

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.

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;
    };
}
23.17.7.3 Class high_resolution_clock  
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.

```cpp
namespace std::chrono {
    class high_resolution_clock {
    public:
        using rep = unspecified;
        using period = ratio<unspecified, unspecified>;
        using duration = chrono::duration<rep, period>;
        using time_point = chrono::time_point<unspecified, duration>;
        static constexpr bool is_steady = unspecified;

        static time_point now() noexcept;
    };
}
```

23.17.8 Header <ctime> synopsis

```cpp
#define NULL see 21.2.3
#define CLOCKS_PER_SEC see below
#define TIME_UTC see below

namespace std {
    using size_t = see 21.2.4;
    using clock_t = see below;
    using time_t = see below;

    struct timespec;
    struct tm;

    clock_t clock();
    double difftime(time_t time1, time_t time0);
    time_t mktime(struct tm* timeptr);
    time_t time(time_t* timer);
    int timespec_get(timespec* ts, int base);
    char* asctime(const struct tm* timeptr);
    char* ctime(const time_t* timer);
    struct tm* gmtime(const time_t* timer);
    struct tm* localtime(const time_t* timer);
    size_t strftime(char* s, size_t maxsize, const char* format, const struct tm* timeptr);
}
```

1 The contents of the header <ctime> are the same as the C standard library header <time.h>.

2 The functions asctime, ctime, gmtime, and localtime are not required to avoid data races (20.5.5.9).

See also: ISO C 7.27

23.18 Class type_index

23.18.1 Header <typeindex> synopsis

```cpp
namespace std {
    class type_index;
    template<class T> struct hash;
    template<> struct hash<type_index>;
}
```

23.18.2 type_index overview

```cpp
namespace std {
    class type_index {
    public:
        type_index(const type_info& rhs) noexcept;
        bool operator==(const type_index& rhs) const noexcept;
    }
```

227) 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.
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.
};

The class type_index provides a simple wrapper for type_info which can be used as an index type in
associative containers (26.4) and in unordered associative containers (26.5).

23.18.3 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;

3 Returns: *target != *rhs.target.

bool operator<(const type_index& rhs) const noexcept;

4 Returns: target->before(*rhs.target).

bool operator<=(const type_index& rhs) const noexcept;

5 Returns: !rhs.target->before(*target).

bool operator>(const type_index& rhs) const noexcept;


bool operator>=(const type_index& rhs) const noexcept;

7 Returns: !target->before(*rhs.target).

size_t hash_code() const noexcept;

8 Returns: target->hash_code().

const char* name() const noexcept;

9 Returns: target->name().

23.18.4 Hash support [type.index.hash]
template<> struct hash<type_index>;

1 For an object index of type type_index, hash<type_index>()(index) shall evaluate to the same
  result as index.hash_code().

23.19 Execution policies [execpol]
23.19.1 In general [execpol.general]

1 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:
    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]

23.19.2 Header <execution> synopsis [execution.syn]

namespace std {

// 23.19.3, 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 {

// 23.19.4, sequenced execution policy
class sequenced_policy;

// 23.19.5, parallel execution policy
class parallel_policy;

// 23.19.6, parallel and unsequenced execution policy
class parallel_unsequenced_policy;

// 23.19.7, execution policy objects
inline constexpr sequenced_policy seq{ unspecified };
inline constexpr parallel_policy par{ unspecified };
inline constexpr parallel_unsequenced_policy par_unseq{ unspecified };
}

23.19.3 Execution policy type trait [execpol.type]

template<class T> struct is Execution_policy { see below };

1 is Execution_policy can be used to detect execution policies for the purpose of excluding function signatures from otherwise ambiguous overload resolution participation.

2 is Execution_policy<T> shall be a UnaryTypeTrait 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]

3 The behavior of a program that adds specializations for is Execution_policy is undefined.

23.19.4 Sequenced execution policy [execpol.seq]

class execution::sequenced_policy { unspecified };

1 The class execution::sequenced_policy is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm’s execution may not be parallelized.

2 During the execution of a parallel algorithm with the execution::sequenced_policy policy, if the invocation of an element access function exits via an uncaught exception, terminate() shall be called.
23.19.5 Parallel execution policy

```cpp
class execution::parallel_policy { unspecified; }
```

1. The class `execution::parallel_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized.

2. During the execution of a parallel algorithm with the `execution::parallel_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

23.19.6 Parallel and unsequenced execution policy

```cpp
class execution::parallel_unsequenced_policy { unspecified; }
```

1. The class `execution::parallel_unsequenced_policy` is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelized and vectorized.

2. During the execution of a parallel algorithm with the `execution::parallel_unsequenced_policy` policy, if the invocation of an element access function exits via an uncaught exception, `terminate()` shall be called.

23.19.7 Execution policy objects

```cpp
inline constexpr execution::sequenced_policy execution::seq{ unspecified; }
inline constexpr execution::parallel_policy execution::par{ unspecified; }
inline constexpr execution::parallel_unsequenced_policy execution::par_unseq{ unspecified; }
```

1. The header `<execution>` declares global objects associated with each type of execution policy.

23.20 Primitive numeric conversions

23.20.1 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
};

// 23.20.2, 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);
```

§ 23.20.1
// 23.20.3, 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, char_format fmt = char_format::general);
from_chars_result from_chars(const char* first, const char* last, double& value, char_format fmt = char_format::general);
from_chars_result from_chars(const char* first, const char* last, long double& value, char_format fmt = char_format::general);

23.20.2 Primitive numeric output conversion  [charconv.to.chars]

All functions named to_chars convert value into a character string by successively filling the range [first, last), where (first, last) is required to be a valid range. If the member ec of the return value is such that the value is equal to the value of a value-initialized errc, the conversion was successful and the member ptr is the one-past-the-end pointer of the characters written. Otherwise, the member ec has the value errc::value_too_large, the member ptr has the value last, and the contents of the range [first, last) are unspecified.

The functions that take a floating-point value but not a precision parameter ensure that the string representation consists of the smallest number of characters such that there is at least one digit before the radix point (if present) and parsing the representation using the corresponding from_chars function recovers value exactly. [ Note: This guarantee applies only if to_chars and from_chars are executed on the same implementation. — end note ] If there are several such representations, the representation with the smallest difference from the floating-point argument value is chosen, resolving any remaining ties using rounding according to round_to_nearest (21.3.3.1).

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.

to_chars_result to_chars(char* first, char* last, see below value, int base = 10);
	nothing.

Remarks: The implementation shall provide overloads for all signed and unsigned integer types and char as the type of the parameter value.

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);

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.

Throws: Nothing.

to_chars_result to_chars(char* first, char* last, float value, char_format fmt);

to_chars_result to_chars(char* first, char* last, double value, char_format fmt);

to_chars_result to_chars(char* first, char* last, long double value, char_format fmt);

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.
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);
```

13  **Requires:** `fmt` has the value of one of the enumerators of `chars_format`.

14  **Effects:** `value` is converted to a string in the style of `printf` in the "C" locale with the given precision.

15  **Throws:** Nothing.

**See also:** ISO C 7.21.6.1

### 23.20.3 Primitive numeric input conversion

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` (21.3.3.1), and the member `ec` is value-initialized.

```c

from_chars_result from_chars(const char* first, const char* last,
                           see below& value, int base = 10);
```

1  **Requires:** `base` has a value between 2 and 36 (inclusive).

2  **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 a minus sign is the only sign that may appear, and only if `value` has a signed type.

3  **Throws:** Nothing.

4  **Remarks:** The implementation shall provide overloads for all signed and unsigned integer types and `char` as the referenced type of the parameter `value`.

```c

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);
```

5  **Requires:** `fmt` has the value of one of the enumerators of `chars_format`.

6  **Effects:** The pattern is the expected form of the subject sequence in the "C" locale, as described for `strtod`, except that

7  (7.1) the only sign that may appear is a minus sign;
7  (7.2) if `fmt` has `chars_format::scientific` set but not `chars_format::fixed`, the otherwise optional exponent part shall appear;
7  (7.3) if `fmt` has `chars_format::fixed` set but not `chars_format::scientific`, the optional exponent part shall not appear; and
7  (7.4) if `fmt` is `chars_format::hex`, the prefix "0x" or "0X" is assumed. [Example: The string 0x123 is parsed to have the value 0 with remaining characters x123. —end example]

In any case, the resulting `value` is one of at most two floating-point values closest to the value of the string matching the pattern.

8  **Throws:** Nothing.
See also: ISO C 7.22.1.3, 7.22.1.4
24 Strings library

24.1 General

1 This Clause describes components for manipulating sequences of any non-array trivial (6.7) type. Such types are called char-like types, and objects of char-like types are called char-like objects or simply characters.

2 The following subclauses describe a character traits class, string classes, and null-terminated sequence utilities, as summarized in Table 53.

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.2 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>24.3 String classes</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>24.4 String view classes</td>
<td>&lt;string_view&gt;</td>
</tr>
<tr>
<td>24.5 Null-terminated sequence utilities</td>
<td>&lt;cstring&gt;</td>
</tr>
</tbody>
</table>

Table 53 — Strings library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.2 Character traits</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>24.3 String classes</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>24.4 String view classes</td>
<td>&lt;string_view&gt;</td>
</tr>
<tr>
<td>24.5 Null-terminated sequence utilities</td>
<td>&lt;cstring&gt;</td>
</tr>
</tbody>
</table>

24.2 Character traits

1 This subclause defines requirements on classes representing character traits, and defines a class template char_traits<charT>, along with four specializations, char_traits<char>, char_traits<char16_t>, char_traits<char32_t>, and char_traits<wchar_t>, that satisfy those requirements.

2 Most classes specified in 24.3 and Clause 30 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.

3 To specialize those templates to generate a string or iostream class to handle a particular character container type charT, that and its related character traits class Traits are passed as a pair of parameters to the string or iostream template as parameters charT and traits. Traits::char_type shall be the same as charT.

4 This subclause specifies a class template, char_traits<charT>, and four explicit specializations of it, char_traits<char>, char_traits<char16_t>, char_traits<char32_t>, and char_traits<wchar_t>, all of which appear in the header <string> and satisfy the requirements below.

24.2.1 Character traits requirements

1 In Table 54, X denotes a Traits class defining types and functions for the character container type charT; c and d denote values of type charT; p and q denote values of type const charT*; s denotes a value of type charT*; 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; state denotes a value of type X::state_type; and r denotes an lvalue of type charT. Operations on Traits shall not throw exceptions.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::char_type</td>
<td>charT</td>
<td>(described in 24.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::int_type</td>
<td>charT</td>
<td>(described in 24.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::off_type</td>
<td>charT</td>
<td>(described in 24.2.2)</td>
<td>compile-time</td>
</tr>
<tr>
<td>X::pos_type</td>
<td>charT</td>
<td>(described in 24.2.2)</td>
<td>compile-time</td>
</tr>
</tbody>
</table>
Table 54 — Character traits requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::state_type</td>
<td>(described in 24.2.2)</td>
<td>compile-time</td>
<td></td>
</tr>
<tr>
<td>X::eq(c,d)</td>
<td>bool</td>
<td><em>Returns:</em> whether c is to be treated as equal to d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::lt(c,d)</td>
<td>bool</td>
<td><em>Returns:</em> 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><em>Returns:</em> 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. <em>Returns</em>: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::copy(s,p,n)</td>
<td>X::char_type*</td>
<td>Requires: p not in [s,s+n). <em>Returns</em>: s. for each i in [0,n), performs X::assign(s[i],p[i]).</td>
<td>linear</td>
</tr>
<tr>
<td>X::assign(r,d)</td>
<td>(not used)</td>
<td>assigns r=d.</td>
<td>constant</td>
</tr>
<tr>
<td>X::assign(s,n,c)</td>
<td>X::char_type*</td>
<td>for each i in [0,n), performs X::assign(s[i],c). <em>Returns</em>: s.</td>
<td>linear</td>
</tr>
<tr>
<td>X::not_eof(e)</td>
<td>int_type</td>
<td><em>Returns</em>: 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><em>Returns</em>: 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><em>Returns</em>: some value e, constrained by the definitions of to_char_type and eq_int_type.</td>
<td>constant</td>
</tr>
</tbody>
</table>
Table 54 — 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::eq_int_type(e, f)</td>
<td>bool</td>
<td><em>Returns:</em> for all c and d,</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X::eq(c, d) is equal to X::eq_</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- int_type(X::to_int_type(c),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>X::to_int_type(d));</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>otherwise, yields true if e and f</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>are both copies of X::eof();</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>otherwise, yields false if one of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>e and f is a copy of X::eof()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>and the other is not; otherwise</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the value is unspecified.</td>
<td></td>
</tr>
</tbody>
</table>

| X::eof()            | X::int_type | *Returns:* a value e such that    | constant   |
|                     |             | X::eq_int_type(e, X::to_int_type(c)) is false for all values c. |            |

2 The class template

```cpp
template<class charT> struct char_traits;
```

shall be provided in the header `<string>` as a basis for explicit specializations.

### 24.2.2 Traits typedefs

```cpp
using char_type = CHAR_T;
```

The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 24.3 and Clause 30.

```cpp
using int_type = INT_T;
```

2 *Requires:* For a certain character container type `char_type`, a related container type `INT_T` shall be a type or class which can represent all of the valid characters converted from the corresponding `char_type` values, as well as an end-of-file value, `eof()`. The type `int_type` represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions.  

```cpp
using off_type = implementation-defined;
```

```cpp
using pos_type = implementation-defined;
```

3 *Requires:* Requirements for `off_type` and `pos_type` are described in 30.2.2 and 30.3.

```cpp
using state_type = STATE_T;
```

4 *Requires:* `state_type` shall meet the requirements of `CopyAssignable` (Table 26), `CopyConstructible` (Table 24), and `DefaultConstructible` (Table 22) types.

### 24.2.3 char_traits specializations

```cpp
namespace std {
    template<> struct char_traits<char>;
    template<> struct char_traits<char16_t>;
    template<> struct char_traits<char32_t>;
    template<> struct char_traits<wchar_t>;
}
```

1 The header `<string>` shall define four specializations of the class template `char_traits`: `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`.

2 The requirements for the members of these specializations are given in 24.2.1.

228) If `eof()` can be held in `char_type` then some iostream operations may give surprising results.
24.2.3.1 struct char_traits<char>

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 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 defined types for int_type, pos_type, off_type, and state_type shall be int, streampos, streamoff, and mbstate_t respectively.

2 The type streampos shall be an implementation-defined type that satisfies the requirements for pos_type in 30.2.2 and 30.3.

3 The type streamoff shall be an implementation-defined type that satisfies the requirements for off_type in 30.2.2 and 30.3.

4 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.

5 The two-argument member assign shall be defined identically to the built-in operator =. The two-argument members eq and lt shall be defined identically to the built-in operators == and < for type unsigned char.

6 The member eof() shall return EOF.

24.2.3.2 struct char_traits<char16_t>

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);
    };

§ 24.2.3.2 663
\[\text{static constexpr int\_type not\_eof(int\_type c) noexcept;}
\text{static constexpr char\_type to\_char\_type(int\_type c) noexcept;}
\text{static constexpr int\_type to\_int\_type(char\_type c) noexcept;}
\text{static constexpr bool eq\_int\_type(int\_type c1, int\_type c2) noexcept;}
\text{static constexpr int\_type eof() noexcept;}
\]

1 The type \text{u16streampos} shall be an implementation-defined type that satisfies the requirements for \text{pos\_type} in 30.2.2 and 30.3.

2 The two-argument members \text{assign}, \text{eq}, and \text{lt} shall be defined identically to the built-in operators =, ==, and < respectively.

3 The member \text{eof()} shall return an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

\[\text{24.2.3.3 struct char\_traits<char32\_t>}
\]

\text{namespace std { }
\text{template<> struct char\_traits<char32\_t> { }
\text{using char\_type = char32\_t;}
\text{using int\_type = uint\_least32\_t;}
\text{using off\_type = streamoff;}
\text{using pos\_type = u32streampos;}
\text{using state\_type = mbstate\_t;}

\text{static constexpr void assign(char\_type& c1, const char\_type& c2) noexcept;}
\text{static constexpr bool eq(char\_type c1, char\_type c2) noexcept;}
\text{static constexpr bool lt(char\_type c1, char\_type c2) noexcept;}
\text{static constexpr int compare(const char\_type* s1, const char\_type* s2, size\_t n);}
\text{static constexpr size\_t length(const char\_type* s);}
\text{static constexpr const char\_type* find(const char\_type* s, size\_t n,}
\text{const char\_type& a);}
\text{static char\_type* move(char\_type* s1, const char\_type* s2, size\_t n);}
\text{static char\_type* copy(char\_type* s1, const char\_type* s2, size\_t n);}
\text{static char\_type* assign(char\_type* s, size\_t n, char\_type a);}

\text{static constexpr int\_type not\_eof(int\_type c) noexcept;}
\text{static constexpr char\_type to\_char\_type(int\_type c) noexcept;}
\text{static constexpr int\_type to\_int\_type(char\_type c) noexcept;}
\text{static constexpr bool eq\_int\_type(int\_type c1, int\_type c2) noexcept;}
\text{static constexpr int\_type eof() noexcept;}
\}
\]
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;
};

The defined types for int_type, pos_type, and state_type shall be wint_t, wstreampos, and mbstate_t respectively.

The type wstreampos shall be an implementation-defined type that satisfies the requirements for pos_type in 30.2.2 and 30.3.

The type mbstate_t is defined in <cwchar> and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

The two-argument members assign, eq, and lt shall be defined identically to the built-in operators =, ==, and < respectively.

The member eof() shall return WEOF.

24.3 String classes

The header <string> defines the basic_string class template for manipulating varying-length sequences of char-like objects and four typedef-names, string, u16string, u32string, and wstring, that name the specializations basic_string<char>, basic_string<char16_t>, basic_string<char32_t>, and basic_string<wchar_t>, respectively.

24.3.1 Header <string> synopsis

#include <initializer_list>

namespace std {

// 24.2, character traits
template<class charT> struct char_traits;
template<> struct char_traits<char>;
template<> struct char_traits<char16_t>;
template<> struct char_traits<char32_t>;
template<> struct char_traits<wchar_t>;

// 24.3.2, 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 operator+(const basic_string& lhs, const basic_string& rhs);
template<class charT, class traits, class Allocator>
basic_string operator+(basic_string& lhs, const basic_string& rhs);

§ 24.3.1
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); 
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+(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 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 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 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);

// 24.3.3.8, swap
void swap(basic_string<charT, traits, Allocator>& lhs,
basic_string<charT, traits, Allocator>& rhs)
noexcept(noexcept(lhs.swap(rhs)));

// 24.3.3.9, inserters and extractors

// 24.3.1 667


```
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
getline(basic_istream<charT, traits>&& is,
basic_string<charT, traits, Allocator>& str);

// basic_string typedef names
using string = basic_string<char>;
using u16string = basic_string<char16_t>;
using u32string = basic_string<char32_t>;
using wstring = basic_string<wchar_t>;

// 24.3.4, 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);
string to_string(long double val);

int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);
float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);
wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

namespace pmr {
    template<class charT, class traits = char_traits<charT>>
    using basic_string = std::basic_string<charT, traits, polymorphic_allocator<charT>>;

    using string = basic_string<char>;
    using u16string = basic_string<char16_t>;
    using u32string = basic_string<char32_t>;
    using wstring = basic_string<wchar_t>;
}

// 24.3.5, hash support
template<class T> struct hash;
template<> struct hash<string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;
```
24.3.2 Class template basic_string

The class template basic_string describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a “string” if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a basic_string object is designated by charT.

The member functions of basic_string use an object of the Allocator class passed as a template parameter to allocate and free storage for the contained char-like objects.229

A basic_string is a contiguous container (26.2.1).

In all cases, size() <= capacity().

The functions described in this Clause can report two kinds of errors, each associated with an exception type:

— a length error is associated with exceptions of type length_error (22.2.5);
— an out-of-range error is associated with exceptions of type out_of_range (22.2.6).

namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_string {
        public:
            // 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 26.2
            using const_iterator = implementation-defined; // see 26.2
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;
            static const size_type npos = -1;

            // 24.3.2.2, construct/copy/destroy
            basic_string() 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());
explicit basic_string(basic_string_view<charT, traits> sv, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<InputIterator>
    basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& a = 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);
basic_string& operator=(basic_string_view<charT, traits> sv);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(initializer_list<charT>);

// 24.3.2.3, 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;

// 24.3.2.4, 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 = 0);
    void shrink_to_fit();
    void clear() noexcept;
    [[nodiscard]] bool empty() const noexcept;

// 24.3.2.5, 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();

// 24.3.2.6, modifiers
    basic_string& operator+=(const basic_string& str);
    basic_string& operator+=(basic_string_view<charT, traits> sv);
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);
basic_string& append(basic_string_view<charT, traits> sv);
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);
basic_string& assign(basic_string_view<charT, traits> sv);
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);

basic_string& insert(size_type pos, basic_string_view<charT, traits> sv);

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 n2);

basic_string& insert(size_type pos, const charT* s);

basic_string& insert(size_type pos, size_type n1, 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);

basic_string& replace(size_type pos1, size_type n1, basic_string_view<charT, traits> sv);

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 n1, const charT* s, size_type n2);

basic_string& replace(size_type pos, size_type n1, const charT* s);

basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);

§ 24.3.2
basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2,
                       basic_string_view<charT, traits> sv);
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, const_iterator, 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);

// 24.3.2.7, 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;

size_type find (basic_string_view<charT, traits> sv, size_type pos = 0) const noexcept;
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 rfind(basic_string_view<charT, traits> sv, size_type pos = npos) const noexcept;
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;
size_type find_first_of(basic_string_view<charT, traits> sv,
                        size_type pos = 0) const noexcept;
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 = npos) const;
size_type find_last_of (basic_string_view<charT, traits> sv,
                        size_type pos = npos) const noexcept;
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;
size_type find_first_not_of(basic_string_view<charT, traits> sv,
                            size_type pos = 0) const noexcept;
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 = npos) const;
size_type find_last_not_of (basic_string_view<charT, traits> sv,
                           size_type pos = npos) const noexcept;
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;

basic_string substr(size_type pos = 0, size_type n = npos) const;
int compare(basic_string_view<charT, traits> sv) const noexcept;
int compare(size_type pos1, size_type n1, basic_string_view<charT, traits> sv) const;


```cpp
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;
int compare(size_type pos1, size_type n1, const basic_string& str,  
    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;

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>;
```

### 24.3.2.1 basic_string general requirements

1. If any operation would cause `size()` to exceed `max_size()`, that operation shall throw an exception object of type `length_error`.

2. If any member function or operator of `basic_string` throws an exception, that function or operator shall have no other effect.

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>` shall use an object of type `Allocator` to allocate and free storage for the contained `charT` objects as needed. The `Allocator` object used shall be obtained as described in 26.2.1. In every specialization `basic_string<charT, traits, Allocator>`, the type `traits::char_type` shall satisfy the character traits requirements (24.2), and the type `traits::char_type` shall name the same type as `charT`.

4. References, pointers, and iterators referring to the elements of a `basic_string` sequence may be invalidated by the following uses of that `basic_string` object:

   - as an argument to any standard library function taking a reference to non-const `basic_string` as an argument.\(^{230}\)
   - Calling non-const member functions, except `operator[]`, `at`, `data`, `front`, `back`, `begin`, `rbegin`, `end`, and `rend`.

### 24.3.2.2 basic_string 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 of class `basic_string`. The postconditions of this function are indicated in Table 55.

- `Effects`: Constructs an object of class `basic_string` as indicated in Table 56. In the second form, `str` is left in a valid state with an unspecified value.

\(^{230}\) For example, as an argument to non-member functions `swap()` (24.3.3.8), `operator>>()` (24.3.3.9), and `getline()` (24.3.3.9), or as an argument to `basic_string::swap()`.

---

\[§ 24.3.2.2\]
Table 55 — basic_string(const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>a non-null pointer that is copyable and can have 0 added to it</td>
</tr>
<tr>
<td>size()</td>
<td>0</td>
</tr>
<tr>
<td>capacity()</td>
<td>an unspecified value</td>
</tr>
</tbody>
</table>

Table 56 — basic_string(const basic_string&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by str.data()</td>
</tr>
<tr>
<td>size()</td>
<td>str.size()</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

`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());`

Throws: out_of_range if `pos > str.size()`.

Effects: Constructs an object of class `basic_string` and determines the effective length `rlen` of the initial string value as `str.size() - pos` in the first form and as the smaller of `str.size() - pos` and `n` in the second form, as indicated in Table 57.

Table 57 — basic_string(const basic_string&, size_type, const Allocator&) and basic_string(const basic_string&, size_type, size_type, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of <code>rlen</code> consequent elements of the string controlled by <code>str</code> beginning at position <code>pos</code></td>
</tr>
<tr>
<td>size()</td>
<td><code>rlen</code></td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

`template<class T>

basic_string(const T& t, size_type pos, size_type n, const Allocator& a = Allocator());`

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);`

Remarks: This constructor shall not participate in overload resolution unless `is_convertible_v<const T&, basic_string_view<charT, traits>>` is true.

`explicit basic_string(basic_string_view<charT, traits> sv, const Allocator& a = Allocator());`

Effects: Same as `basic_string(sv.data(), sv.size(), a).`

`basic_string(const charT* s, size_type n, const Allocator& a = Allocator());`

Requires: `s` points to an array of at least `n` elements of `charT`.

Effects: Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `n` whose first element is designated by `s`, as indicated in Table 58.

`basic_string(const charT* s, const Allocator& a = Allocator());`

Requires: `s` points to an array of at least `traits::length(s) + 1` elements of `charT`. 

§ 24.3.2.2
Table 58 — basic_string(const charT*, size_type, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by s</td>
</tr>
<tr>
<td>size()</td>
<td>n</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

Effects: Constructs an object of class basic_string and determines its initial string value from the array of charT of length traits::length(s) whose first element is designated by s, as indicated in Table 59.

Table 59 — basic_string(const charT*, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by s</td>
</tr>
<tr>
<td>size()</td>
<td>traits::length(s)</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

basic_string(size_type n, charT c, const Allocator& a = Allocator());

Requires: n < npos.

Effects: Constructs an object of class basic_string and determines its initial string value by repeating the char-like object c for all n elements, as indicated in Table 60.

Table 60 — basic_string(size_t, charT, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated array of n elements, each storing the initial value c</td>
</tr>
<tr>
<td>size()</td>
<td>n</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

template<class InputIterator>
basic_string(InputIterator begin, InputIterator end, const Allocator& a = Allocator());

Effects: If InputIterator is an integral type, equivalent to:

basic_string(static_cast<size_type>(begin), static_cast<value_type>(end), a);

Otherwise constructs a string from the values in the range [begin, end), as indicated in the Sequence Requirements table (see 26.2.3).

basic_string(initializer_list<charT> il, const Allocator& a = Allocator());

Effects: Same as basic_string(il.begin(), il.end(), a).

basic_string(const basic_string& str, const Allocator& alloc);
basic_string(basic_string&& str, const Allocator& alloc);

Effects: Constructs an object of class basic_string as indicated in Table 61. The stored allocator is constructed from alloc. In the second form, str is left in a valid state with an unspecified value.

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>,
Table 61 — basic_string(const basic_string&, const Allocator&) and basic_string(basic_string&&, const Allocator&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by the original value of str.data().</td>
</tr>
<tr>
<td>size()</td>
<td>the original value of str.size()</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
<tr>
<td>get_allocator()</td>
<td>alloc</td>
</tr>
</tbody>
</table>

Remarks: Shall not participate in overload resolution if InputIterator is a type that does not qualify as an input iterator, or if Allocator is a type that does not qualify as an allocator (26.2.1).

basic_string& operator=(const basic_string& str);

Effects: If *this and str are not the same object, modifies *this as shown in Table 62.
If *this and str are the same object, the member has no effect.

Returns: *this.

Table 62 — operator=(const basic_string&) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data()</td>
<td>points at the first element of an allocated copy of the array whose first element is pointed at by str.data()</td>
</tr>
<tr>
<td>size()</td>
<td>str.size()</td>
</tr>
<tr>
<td>capacity()</td>
<td>a value at least as large as size()</td>
</tr>
</tbody>
</table>

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 (26.2), except that iterators, pointers and references may be invalidated.

Returns: *this.

basic_string& operator=(basic_string_view<charT, traits> sv);

Effects: Equivalent to: return assign(sv);

basic_string& operator=(const charT* s);

Returns: *this = basic_string(s).
Remarks: Uses traits::length().

basic_string& operator=(charT c);

Returns: *this = basic_string(1, c).

basic_string& operator=(initializer_list<charT> il);

Effects: As if by: *this = basic_string(il);

Returns: *this.

24.3.2.3 basic_string iterator support [string.iterators]

iterator begin() noexcept;
const_iterator begin() const noexcept;
const_iterator cbegin() const noexcept;
1  
Returns: An iterator referring to the first character in the string.

iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cend() const noexcept;
2  
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;
3  
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;
4  
Returns: An iterator which is semantically equivalent to reverse_iterator(begin()).

24.3.2.4 basic_string capacity  

size_type size() const noexcept;
1  
Returns: A count of the number of char-like objects currently in the string.

Complexity: Constant time.

size_type length() const noexcept;
3  
Returns: size().

size_type max_size() const noexcept;
4  
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);
6  
Throws: length_error if n > max_size().

Effects: Alters the length of the string designated by *this as follows:

(7.1) — If n <= size(), the function replaces the string designated by *this with a string of length n whose elements are a copy of the initial elements of the original string designated by *this.

(7.2) — If n > size(), the function replaces the string designated by *this with a string of length n whose first size() elements are a copy of the original string designated by *this, and whose remaining elements are all initialized to c.

void resize(size_type n);
8  
Effects: As if by resize(n, charT()).

size_type capacity() const noexcept;
9  
Returns: The size of the allocated storage in the string.

void reserve(size_type res_arg=0);
10

The member function reserve() is a directive that informs a basic_string object of a planned change in size, so that it can manage the storage allocation accordingly.

Effects: After reserve(), capacity() is greater or equal to the argument of reserve. [Note: Calling reserve() with a res_arg argument less than capacity() is in effect a non-binding shrink request. A call with res_arg <= size() is in effect a non-binding shrink-to-fit request. — end note]

11  
Throws: length_error if res_arg > max_size().231

231) reserve() uses allocator_traits<Allocator>::allocate() which may throw an appropriate exception.
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: 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. If no reallocation happens, they remain valid.

void clear() noexcept;

Effects: Behaves as if the function calls:

erase(begin(), end());

[[nodiscard]] bool empty() const noexcept;

Returns: size() == 0.

### 24.3.2.5 basic_string element access

#### 24.3.2.5.1 basic_string::operator[]

const_reference operator[](size_type pos) const;

reference operator[](size_type pos);

Requires: 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();

Requires: !empty().

Effects: Equivalent to: return operator[](0);

const charT& back() const;

charT& back();

Requires: !empty().

Effects: Equivalent to: return operator[](size() - 1);

### 24.3.2.6 basic_string modifiers

#### 24.3.2.6.1 basic_string::operator+=

basic_string& operator+=(const basic_string& str);

Effects: Calls append(str).

Returns: *this.

basic_string& operator+=(basic_string_view<charT, traits> sv);

Effects: Calls append(sv).

Returns: *this.

basic_string& operator+=(const charT* s);

Effects: Calls append(s).

Returns: *this.
basic_string& operator+=(charT c);

  Effects: Calls push_back(c);

  Returns: *this.

basic_string& operator+=(initializer_list<charT> il);

  Effects: Calls append(il).

  Returns: *this.

24.3.2.6.2 basic_string::append

  [string.append]

basic_string& append(const basic_string& str);

  Effects: Calls append(str.data(), str.size()).

  Returns: *this.

basic_string& append(const basic_string& str, size_type pos, size_type n = npos);

  Throws: out_of_range if pos > str.size().

  Effects: Determines the effective length rlen of the string to append as the smaller of n and str.size() - pos and calls append(str.data() + pos, rlen).

  Returns: *this.

basic_string& append(basic_string_view<charT, traits> sv);

  Effects: Equivalent to: return append(sv.data(), sv.size());

template<class T>
basic_string& append(const T& t, size_type pos, size_type n = npos);

  Throws: out_of_range if pos > sv.size().

  Effects: Creates a variable, sv, as if by basic_string_view<charT, traits> sv = t. Determines the effective length rlen of the string to append as the smaller of n and sv.size() - pos and calls append(sv.data() + pos, rlen).

  Remarks: This function shall not participate in overload resolution unless is_convertible_v<const T& basic_string_view<charT, traits>> is true and is_convertible_v<const T&, const charT*> is false.

  Returns: *this.

basic_string& append(const charT* s, size_type n);

  Requires: s points to an array of at least n elements of charT.

  Throws: length_error if size() + n > max_size().

  Effects: The function replaces the string controlled by *this with a string of length size() + n whose first size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial n elements of s.

  Returns: *this.

basic_string& append(const charT* s);

  Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

  Effects: Calls append(s, traits::length(s)).

  Returns: *this.

basic_string& append(size_type n, charT c);

  Effects: Equivalent to: return append(basic_string(n, c));

template<class InputIterator>
basic_string& append(InputIterator first, InputIterator last);

  Requires: [first, last) is a valid range.

  Effects: Equivalent to: return append(basic_string(first, last, get_allocator()));
basic_string& append(initializer_list<charT> il);

   Effects: Calls append(il.begin(), il.size()).
   Returns: *this.

void push_back(charT c);

   Effects: Equivalent to append(static_cast<size_type>(1), c).

24.3.2.6.3 basic_string::assign

basic_string& assign(const basic_string& str);

   Effects: Equivalent to: return *this = str;

basic_string& assign(basic_string& str)
   noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
   allocator_traits<Allocator>::is_always_equal::value);

   Effects: Equivalent to: return *this = std::move(str);

basic_string& assign(const basic_string& str, size_type pos, size_type n = npos);

   Throws: out_of_range if pos > str.size().
   Effects: Determines the effective length rlen of the string to assign as the smaller of n and str.size() - pos and calls assign(str.data() + pos, rlen).
   Returns: *this.

basic_string& assign(basic_string_view<charT, traits> sv);

   Effects: Equivalent to: return assign(sv.data(), sv.size());

template<class T>
basic_string& assign(const T& t, size_type pos, size_type n = npos);

   Throws: out_of_range if pos > sv.size().
   Effects: Creates a variable, sv, as if by basic_string_view<charT, traits> sv = t. Determines the effective length rlen of the string to assign as the smaller of n and sv.size() - pos and calls assign(sv.data() + pos, rlen).
   Remarks: This function shall not participate in overload resolution unless is_convertible_v<const T&, basic_string_view<charT, traits>> is true and is_convertible_v<const T&, const charT*> is false.
   Returns: *this.

basic_string& assign(const charT* s, size_type n);

   Requires: s points to an array of at least n elements of charT.
   Throws: length_error if n > max_size().
   Effects: Replaces the string controlled by *this with a string of length n whose elements are a copy of those pointed to by s.
   Returns: *this.

basic_string& assign(const charT* s);

   Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
   Effects: Calls assign(s, traits::length(s)).
   Returns: *this.

basic_string& assign(initializer_list<charT> il);

   Effects: Calls assign(il.begin(), il.size()).
   Returns: *this.

basic_string& assign(size_type n, charT c);

   Effects: Equivalent to: return assign(basic_string(n, c));
template<class InputIterator>
   basic_string& assign(InputIterator first, InputIterator last);

   Effects: Equivalent to: return assign(basic_string(first, last, get_allocator()));

24.3.2.6.4 basic_string::insert [string.insert]

   basic_string& insert(size_type pos, const basic_string& str);

   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);

   Throws: out_of_range if pos1 > size() or pos2 > str.size().

   Effects: Determines the effective length rlen of the string to insert as the smaller of n and str.size() - pos2 and calls insert(pos1, str.data() + pos2, rlen).

   Returns: *this.

   basic_string& insert(size_type pos, basic_string_view<charT, traits> sv);

   Effects: Equivalent to: return insert(pos, sv.data(), sv.size());

   template<class T>
   basic_string& insert(size_type pos1, const T& t, size_type pos2, size_type n = npos);

   Throws: out_of_range if pos1 > size() or pos2 > sv.size().

   Effects: Creates a variable, sv, as if by basic_string_view<charT, traits> sv = t. Determines the effective length rlen of the string to assign as the smaller of n and sv.size() - pos2 and calls insert(pos1, sv.data() + pos2, rlen).

   Remarks: This function shall not participate in overload resolution unless is_convertible_v<const T&, basic_string_view<charT, traits>> is true and is_convertible_v<const T&, const charT*> is false.

   Returns: *this.

   basic_string& insert(size_type pos, const charT* s, size_type n);

   Requires: s points to an array of at least n elements of charT.

   Throws: out_of_range if pos > size() or length_error if size() + n > max_size().

   Effects: Replaces the string controlled by *this with a string of length size() + n whose first pos elements are a copy of the initial elements of the original string controlled by *this and whose next n elements are a copy of the elements in s and whose remaining elements are a copy of the remaining elements of the original string controlled by *this.

   Returns: *this.

   basic_string& insert(size_type pos, const charT* s);

   Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

   Effects: Equivalent to: return insert(pos, s, traits::length(s));

   basic_string& insert(size_type pos, size_type n, charT c);

   Effects: Equivalent to: return insert(pos, basic_string(n, c));

   iterator insert(const_iterator p, charT c);

   Requires: p is a valid iterator on *this.

   Effects: Inserts a copy of c before the character referred to by p.

   Returns: An iterator which refers to the copy of the inserted character.

   iterator insert(const_iterator p, size_type n, charT c);

   Requires: p is a valid iterator on *this.

   Effects: Inserts n copies of c before the character referred to by p.

   Returns: An iterator which refers to the copy of the first inserted character, or p if n == 0.
template<class InputIterator>
iterator insert(const_iterator p, InputIterator first, InputIterator last);

Requires: p is a valid iterator on *this. [first, last) is a valid range.
Effects: Equivalent to insert(p - begin(), basic_string(first, last, get_allocator())).
Returns: An iterator which refers to the copy of the first inserted character, or p if first == last.

iterator insert(const_iterator p, initializer_list<charT> il);

Effects: As if by insert(p, il.begin(), il.end()).
Returns: An iterator which refers to the copy of the first inserted character, or p if il is empty.

24.3.2.6.5 basic_string::erase [string.erase]

basic_string& erase(size_type pos = 0, size_type n = npos);

1 Throws: out_of_range if pos > size().
2 Effects: Determines the effective length xlen of the string to be removed as the smaller of n and size() - pos.
   The function then replaces the string controlled by *this with a string of length size() - xlen whose first pos elements are a copy of the initial elements of the original string controlled by *this, and whose remaining elements are a copy of the elements of the original string controlled by *this beginning at position pos + xlen.
3 Returns: *this.

iterator erase(const_iterator p);

5 Throws: Nothing.
6 Effects: Removes the character referred to by p.
7 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);

8 Requires: first and last are valid iterators on *this, defining a range [first, last).
9 Throws: Nothing.
10 Effects: Removes the characters in the range [first, last).
11 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.

void pop_back();

12 Requires: !empty().
13 Throws: Nothing.
14 Effects: Equivalent to erase(size() - 1, 1).

24.3.2.6.6 basic_string::replace [string.replace]

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);

1 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);

2 Throws: out_of_range if pos1 > size() or pos2 > str.size().
3 Effects: Determines the effective length rlen of the string to be inserted as the smaller of n2 and str.size() - pos2 and calls replace(pos1, n1, str.data() + pos2, rlen).
4 Returns: *this.
basic_string& replace(size_type pos1, size_type n1,
    basic_string_view<charT, traits> sv);

 Effects: Equivalent to: 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);

 Throws: out_of_range if pos1 > size() or pos2 > sv.size().

 Effects: Creates a variable, sv, as if by basic_string_view<charT, traits> sv = t. Determines
    the effective length rlen of the string to be inserted as the smaller of n2 and sv.size() - pos2
    and calls replace(pos1, n1, sv.data() + pos2, rlen).

 Remarks: This function shall not participate in overload resolution unless
    is_convertible_v<const T&, basic_string_view<charT, traits>> is true and
    is_convertible_v<const T&, const charT*> is false.

 Returns: *this.

basic_string& replace(size_type pos1, size_type n1, const charT* s, size_type n2);

 Requires: s points to an array of at least n2 elements of charT.

 Throws: out_of_range if pos1 > size() or length_error if the length of the resulting string would
    exceed max_size() (see below).

 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 string controlled by *this with a string of length size() - xlen + n2
    whose first pos1 elements are a copy of the initial elements of the original string controlled by
    *this, whose next n2 elements are a copy of the initial n2 elements of s, and whose remaining
    elements are a copy of the elements of the original string controlled by *this beginning at
    position pos + xlen.

 Returns: *this.

basic_string& replace(size_type pos, size_type n, const charT* s);

 Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

 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);

 Effects: Equivalent to: return replace(pos1, n1, basic_string(n2, c));

basic_string& replace(const_iterator i1, const_iterator i2, const basic_string& str);

 Requires: [begin(), i1) and [i1, i2) are valid ranges.

 Effects: Calls replace(i1 - begin(), i2 - i1, str).

 Returns: *this.

basic_string& replace(const_iterator i1, const_iterator i2, basic_string_view<charT, traits> sv);

 Requires: [begin(), i1) and [i1, i2) are valid ranges.

 Effects: Calls replace(i1 - begin(), i2 - i1, sv).

 Returns: *this.

basic_string& replace(const_iterator i1, const_iterator i2, const charT* s, size_type n);

 Requires: [begin(), i1) and [i1, i2) are valid ranges and s points to an array of at least n
    elements of charT.

 Effects: Calls replace(i1 - begin(), i2 - i1, s, n).

 Returns: *this.

basic_string& replace(const_iterator i1, const_iterator i2, const charT* s);

 Requires: [begin(), i1) and [i1, i2) are valid ranges and s points to an array of at least
    traits::length(s) + 1 elements of charT.
27. **Effects:** Calls \texttt{replace(i1 - begin(), i2 - i1, s, traits::length(s))}.

28. **Returns:** \*this.

29. \texttt{basic_string& replace(const_iterator i1, const_iterator i2, size_type n, charT c);}  
   **Requires:** \texttt{[begin(), i1)} and \texttt{[i1, i2)} are valid ranges.

30. **Effects:** Calls \texttt{replace(i1 - begin(), i2 - i1, basic_string(n, c))}.

31. **Returns:** \*this.

32. \texttt{template<class InputIterator>}  
   \texttt{basic_string& replace(const_iterator i1, const_iterator i2, InputIterator j1, InputIterator j2);}  
   **Requires:** \texttt{[begin(), i1)}, \texttt{[i1, i2)} and \texttt{[j1, j2)} are valid ranges.

33. **Effects:** Calls \texttt{replace(i1 - begin(), i2 - i1, basic_string(j1, j2, get_allocator()))}.

34. **Returns:** \*this.

35. \texttt{basic_string& replace(const_iterator i1, const_iterator i2, initializer_list<charT> il);}  
   **Requires:** \texttt{[begin(), i1)} and \texttt{[i1, i2)} are valid ranges.

36. **Effects:** Calls \texttt{replace(i1 - begin(), i2 - i1, il.begin(), il.size())}.

37. **Returns:** \*this.

24.3.2.6.7 **basic_string::copy**  
   \texttt{size_type copy(charT* s, size_type n, size_type pos = 0) const;}  
   **1** Let \texttt{rlen} be the smaller of \texttt{n} and \texttt{size()} - \texttt{pos}.

   **2** Throw: out_of_range if \texttt{pos > size()}.

   **3** Requires: \texttt{[s, s + rlen)} is a valid range.

   **4** Effects: Equivalent to \texttt{traits::copy(s, data() + pos, rlen)}. [Note: This does not terminate \texttt{s} with a null object. —end note]

   **5** Returns: \texttt{rlen}.

24.3.2.6.8 **basic_string::swap**  
   \texttt{void swap(basic_string& s);}  
   **1** Postconditions: \*this contains the same sequence of characters that was in \texttt{s}, \texttt{s} contains the same sequence of characters that was in \*this.

   **2** Throws: Nothing.

   **3** Complexity: Constant time.

24.3.2.7 **basic_string string operations**  
24.3.2.7.1 **basic_string accessors**  

\texttt{const charT* c_str() const noexcept;}  
\texttt{const charT* data() const noexcept;}  

**1** Returns: A pointer \texttt{p} such that \texttt{p + i == &operator[](i)} for each \texttt{i} in \texttt{[0, size()]}.  
**2** Complexity: Constant time.  
**3** Requires: The program shall not alter any of the values stored in the character array.

\texttt{charT* data() noexcept;}  
**4** Returns: A pointer \texttt{p} such that \texttt{p + i == &operator[](i)} for each \texttt{i} in \texttt{[0, size()]}.  
**5** Complexity: Constant time.  
**6** Requires: The program shall not alter the value stored at \texttt{p + size()}.  

§ 24.3.2.7.1 684
operator basic_string_view<charT, traits>() const noexcept;

Effects: Equivalent to: return basic_string_view<charT, traits>(data(), size());

allocator_type get_allocator() const noexcept;

Returns: A copy of the Allocator object used to construct the string or, if that allocator has been replaced, a copy of the most recent replacement.

24.3.2.7.2 basic_string::find

std::size_type find(basic_string_view<charT, traits> sv, std::size_type pos = 0) const noexcept;

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions hold:

1. pos <= xpos and xpos + sv.size() <= size();
2. traits::eq(at(xpos + I), sv.at(I)) for all elements I of the data referenced by sv.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

size_type find(const basic_string& str, std::size_type pos = 0) const noexcept;

Effects: Equivalent to: return find(basic_string_view<charT, traits>(str), pos);

size_type find(const charT* s, std::size_type pos, std::size_type n) const;

Returns: find(basic_string_view<charT, traits>(s, n), pos).

size_type find(charT c, std::size_type pos = 0) const;

Returns: find(basic_string(1, c), pos).

24.3.2.7.3 basic_string::rfind

std::size_type rfind(basic_string_view<charT, traits> sv, std::size_type pos = std::npos) const noexcept;

Effects: Determines the highest position xpos, if possible, such that both of the following conditions hold:

1. xpos <= pos and xpos + sv.size() <= size();
2. traits::eq(at(xpos + I), sv.at(I)) for all elements I of the data referenced by sv.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

size_type rfind(const basic_string& str, std::size_type pos = std::npos) const noexcept;

Effects: Equivalent to: return rfind(basic_string_view<charT, traits>(str), pos);

size_type rfind(const charT* s, std::size_type pos, std::size_type n) const;

Returns: rfind(basic_string_view<charT, traits>(s, n), pos).

size_type rfind(charT c, std::size_type pos = std::npos) const;

Returns: rfind(basic_string(1, c), pos).

24.3.2.7.4 basic_string::find_first_of

std::size_type find_first_of(basic_string_view<charT, traits> sv, std::size_type pos = 0) const noexcept;

Effects: Determines the lowest position xpos, if possible, such that both of the following conditions hold:

§ 24.3.2.7.4 685
size_type find_first_of(const basic_string& str, size_type pos = 0) const noexcept;
3 Effects: Equivalent to: return find_first_of(basic_string_view<charT, traits>(str), pos);

size_type find_first_of(const charT* s, size_type pos, size_type n) const;
4 Returns: find_first_of(basic_string_view<charT, traits>(s, n), pos).

size_type find_first_of(charT c, size_type pos = 0) const;
5 Returns: find_first_of(basic_string(1, c), pos).

24.3.2.7.5 basic_string::find_last_of

size_type find_last_of(basic_string_view<charT, traits> sv, size_type pos = npos) const noexcept;
1 Effects: Determines the highest position xpos, if possible, such that both of the following conditions hold:

(1.1) — pos <= xpos and xpos < size();
(1.2) — traits::eq(at(xpos), sv.at(I)) for some element I of the data referenced by sv.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

size_type find_last_of(const basic_string& str, size_type pos = npos) const noexcept;
3 Effects: Equivalent to: return find_last_of(basic_string_view<charT, traits>(str), pos);

size_type find_last_of(const charT* s, size_type pos, size_type n) const;
4 Returns: find_last_of(basic_string_view<charT, traits>(s, n), pos).

size_type find_last_of(const charT* s, size_type pos = npos) const;
5 Requires: s points to an array of at least traits::length(s) + 1 elements of charT.

size_type find_last_of(charT c, size_type pos = npos) const;
6 Returns: find_last_of(basic_string(1, c), pos).

24.3.2.7.6 basic_string::find_first_not_of

size_type find_first_not_of(basic_string_view<charT, traits> sv, size_type pos = 0) const noexcept;
1 Effects: Determines the lowest position xpos, if possible, such that both of the following conditions hold:

(1.1) — pos <= xpos and xpos < size();
(1.2) — traits::eq(at(xpos), sv.at(I)) for no element I of the data referenced by sv.

Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

size_type find_first_not_of(const basic_string& str, size_type pos = 0) const noexcept;
3 Effects: Equivalent to: return find_first_not_of(basic_string_view<charT, traits>(str), pos);

size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
4 Returns: find_first_not_of(basic_string_view<charT, traits>(s, n), pos).
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
Returns: find_first_not_of(basic_string_view<charT, traits>(s), pos).

size_type find_first_not_of(charT c, size_type pos = 0) const;
Returns: find_first_not_of(basic_string(1, c), pos).

24.3.2.7.7 basic_string::find_last_not_of

size_type find_last_not_of(basic_string_view<charT, traits> sv, size_type pos = npos) const noexcept;
Effects: Determines the highest position xpos, if possible, such that both of the following conditions hold:
— xpos <= pos and xpos < size();
— traits::eq(at(xpos), sv.at(I)) for no element I of the data referenced by sv.
Returns: xpos if the function can determine such a value for xpos. Otherwise, returns npos.

size_type find_last_not_of(const basic_string& str, size_type pos = npos) const noexcept;
Effects: Equivalent to:
return find_last_not_of(basic_string_view<charT, traits>(str), pos);

size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
Returns: find_last_not_of(basic_string_view<charT, traits>(s, n), pos).

size_type find_last_not_of(const charT* s, size_type pos = npos) const;
Requires: s points to an array of at least traits::length(s) + 1 elements of charT.
Returns: find_last_not_of(basic_string_view<charT, traits>(s), pos).

size_type find_last_not_of(charT c, size_type pos = npos) const;
Returns: find_last_not_of(basic_string(1, c), pos).

24.3.2.7.8 basic_string::substr

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).

24.3.2.7.9 basic_string::compare

int compare(basic_string_view<charT, traits> sv) const noexcept;
Effects: Determines the effective length rlen of the strings to compare as the smaller of size() and sv.size(). The function then compares the two strings by calling traits::compare(data(), sv.data(), rlen).
Returns: The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 63.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size() &lt; sv.size()</td>
<td>&lt; 0</td>
</tr>
<tr>
<td>size() == sv.size()</td>
<td>0</td>
</tr>
<tr>
<td>size() &gt; sv.size()</td>
<td>&gt; 0</td>
</tr>
</tbody>
</table>
int compare(size_type pos1, size_type n1, basic_string_view<charT, traits> sv) const;

Effects: Equivalent to:

   return basic_string_view<charT, traits>(data(), size()).substr(pos1, n1).compare(sv);

template<class T>
int compare(size_type pos1, size_type n1, const T& t, size_type pos2, size_type n2 = npos) const;

Effects: Equivalent to:

   basic_string_view<charT, traits> sv = t;
   return basic_string_view<charT, traits>(data(), size()).substr(pos1, n1).compare(sv.substr(pos2, n2));

Remarks: This function shall not participate in overload resolution unless is_convertible_v<const T&,
   basic_string_view<charT, traits>> is true and is_convertible_v<const T&, const charT*> is false.

int compare(const basic_string& str) const noexcept;

Effects: Equivalent to:

   return compare(basic_string_view<charT, traits>(str));

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));

int compare(size_type pos1, size_type n1, const basic_string& str,
   size_type pos2, size_type n2 = npos) const;

Effects: Equivalent to:

   return compare(pos1, n1, basic_string_view<charT, traits>(str), pos2, n2);

int compare(const charT* s) const;

Returns: compare(basic_string(s)).

int compare(size_type pos, size_type n1, const charT* s) const;

Returns: basic_string(*this, pos, n1).compare(basic_string(s, n2)).

24.3.2.7.10 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);

24.3.2.7.11 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);

24.3.3 basic_string non-member functions [string.nonmembers]

24.3.3.1 operator+ [string.op+]

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);

Returns: basic_string<charT, traits, Allocator>(lhs).append(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);

Returns: std::move(lhs.append(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);

Returns: std::move(rhs.insert(0, lhs)).

Returns: std::move(lhs.append(rhs)). [Note: Or equivalently, std::move(rhs.insert(0, lhs)).
—end note]

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: basic_string<charT, traits, Allocator>(lhs) + rhs.

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(const charT* lhs, basic_string<charT, traits, Allocator>&& rhs);

Returns: std::move(rhs.insert(0, lhs)).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(charT lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: basic_string<charT, traits, Allocator>(1, lhs) + rhs.

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(charT lhs, basic_string<charT, traits, Allocator>&& rhs);

Returns: std::move(rhs.insert(0, 1, lhs)).

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Returns: lhs + basic_string<charT, traits, Allocator>(rhs).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(basic_string<charT, traits, Allocator>&& lhs, const charT* rhs);

Returns: std::move(lhs.append(rhs)).

Remarks: Uses traits::length().

template<class charT, class traits, class Allocator>

basic_string<charT, traits, Allocator>
operator+(const basic_string<charT, traits, Allocator>& lhs, charT rhs);

Returns: lhs + basic_string<charT, traits, Allocator>(1, rhs).
template<class charT, class traits, class Allocator>
    basic_string<charT, traits, Allocator>
    operator+(basic_string<charT, traits, Allocator>&& lhs, charT rhs);

Returns: std::move(lhs.append(1, rhs)).

24.3.3.2 operator==

 template<class charT, class traits, class Allocator>
 bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs) noexcept;

Returns: lhs.compare(rhs) == 0.

 template<class charT, class traits, class Allocator>
 bool operator==(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: rhs == lhs.

 template<class charT, class traits, class Allocator>
 bool operator==(const basic_string<charT, traits, Allocator>& lhs,
                const charT* rhs);

Requires: rhs points to an array of at least traits::length(rhs) + 1 elements of charT.

Returns: lhs.compare(rhs) == 0.

24.3.3.3 operator!=

 template<class charT, class traits, class Allocator>
 bool operator!=(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs) noexcept;

Returns: !(lhs == rhs).

 template<class charT, class traits, class Allocator>
 bool operator!=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: rhs != lhs.

 template<class charT, class traits, class Allocator>
 bool operator!=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Requires: rhs points to an array of at least traits::length(rhs) + 1 elements of charT.

Returns: lhs.compare(rhs) != 0.

24.3.3.4 operator<

 template<class charT, class traits, class Allocator>
 bool operator<(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs) noexcept;

Returns: lhs.compare(rhs) < 0.

 template<class charT, class traits, class Allocator>
 bool operator<(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: rhs.compare(lhs) > 0.

 template<class charT, class traits, class Allocator>
 bool operator<(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Returns: lhs.compare(rhs) < 0.

24.3.3.5 operator>

 template<class charT, class traits, class Allocator>
 bool operator>(const basic_string<charT, traits, Allocator>& lhs,
                const basic_string<charT, traits, Allocator>& rhs) noexcept;

Returns: lhs.compare(rhs) > 0.

§ 24.3.3.5
template<class charT, class traits, class Allocator>
  bool operator>(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: rhs.compare(lhs) < 0.

template<class charT, class traits, class Allocator>
  bool operator>(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Returns: lhs.compare(rhs) > 0.

24.3.3.6 operator<=

template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: rhs.compare(lhs) >= 0.

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Returns: lhs.compare(rhs) <= 0.

24.3.3.7 operator>=

template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs, const basic_string<charT, traits, Allocator>& rhs);

Returns: rhs.compare(lhs) <= 0.

template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT, traits, Allocator>& lhs, const charT* rhs);

Returns: lhs.compare(rhs) >= 0.

24.3.3.8 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).

24.3.3.9 Inserters and extractors

template<class charT, class traits, class Allocator>
  basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& is,
                                                basic_string<charT, traits, Allocator>&& str);

Effects: Behaves as a formatted input function (30.7.4.2.1). 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:

- n characters are stored;
- end-of-file occurs on the input sequence;
- 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 (30.5.5.4)`.

Returns: `is`.

```cpp
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);`

```cpp
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
gline(basic_istream<charT, traits>&& is,
    basic_string<charT, traits, Allocator>& str,
    charT delim);
```

Effects: Behaves as an unformatted input function (30.7.4.3), except that it does not affect the value returned by subsequent calls to `basic_istream<>::gcount()`. After constructing a `sentry` object, if the sentry converts to `true`, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)` until any of the following occurs:

1. **end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).**
2. **traits::eq(c, delim) for the next available input character `c` (in which case, `c` is extracted but not appended) (30.5.5.4)**
3. **str.max_size() characters are stored (in which case, the function calls `is.setstate(ios_base::failbit)` (30.5.5.4)**

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 (30.5.5.4)`.

Returns: `is`.

```cpp
template<class charT, class traits, class Allocator>
basic_istream<charT, traits>&
gline(basic_istream<charT, traits>& is,
    basic_string<charT, traits, Allocator>& str);
```

Returns: `getline(is, str, is.widen(\'\n\'))`.

### 24.3.4 Numeric conversions

```cpp
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, strtol, or strtoull reports that no conversion could be performed. Throws out_of_range if strtol, strtoul, strtol 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);
string to_string(long double val);

Returns: Each function returns a string object holding the character representation of the value of its argument that would be generated by calling sprintf(buf, fmt, val) with a format specifier of "%d", "%u", "%ld", "%lu", "%lld", "%llu", "%f", "%F", or "%Lf", respectively, where buf designates an internal character buffer of sufficient size.

int stoi(const wstring& str, size_t* idx = nullptr, int base = 10);
long stol(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long stoul(const wstring& str, size_t* idx = nullptr, int base = 10);
long long stoll(const wstring& str, size_t* idx = nullptr, int base = 10);
unsigned long long stoull(const wstring& str, size_t* idx = nullptr, int base = 10);

Effects: The first two functions call wcstol(str.c_str(), ptr, base), and the last three functions call wcstoul(str.c_str(), ptr, base), wcstoll(str.c_str(), ptr, base), and wcstoull(str.c_str(), ptr, base), respectively. Each function returns the converted result, if any. The argument ptr designates a pointer to an object internal to the function that is used to determine what to store at *idx. If the function does not throw an exception and idx != 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.

float stof(const wstring& str, size_t* idx = nullptr);
double stod(const wstring& str, size_t* idx = nullptr);
long double stold(const wstring& str, size_t* idx = nullptr);

Effects: These functions call wcstof(str.c_str(), ptr), wcstod(str.c_str(), ptr), and wcstold(str.c_str(), ptr), respectively. Each function returns the converted result, if any. The argument ptr designates a pointer to an object internal to the function that is used to determine what to store at *idx. If the function does not throw an exception and idx != 0, the function stores in *idx the index of the first unconverted element of str.
Returns: The converted result.

Throws: invalid_argument if wcstof, wcstod, or wcstold reports that no conversion could be performed. Throws out_of_range if wcstof, wcstod, or wcstold sets errno to ERANGE.

wstring to_wstring(int val);
wstring to_wstring(unsigned val);
wstring to_wstring(long val);
wstring to_wstring(unsigned long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(float val);
wstring to_wstring(double val);
wstring to_wstring(long double val);

Returns: Each function returns a wstring object holding the character representation of the value of its argument that would be generated by calling swprintf(buf, buffsz, fmt, val) with a format specifier of L"%d", L"%u", L"%ld", L"%lu", L"%lld", L"%llu", L"%lf", L"%f", or L"%Lf", respectively, where buf designates an internal character buffer of sufficient size buffsz.

24.3.5 Hash support

template<> struct hash<string>;
template<> struct hash<u16string>;
template<> struct hash<u32string>;
template<> struct hash<wstring>;
template<> struct hash<pmr::string>;
template<> struct hash<pmr::u16string>;
template<> struct hash<pmr::u32string>;
template<> struct hash<pmr::wstring>;

If S is one of these string types, SV is the corresponding string view type, and s is an object of type S, then hash<S>()(s) == hash<SV>()(SV(s)).

24.3.6 Suffix for basic_string literals

string operator"s(const char* str, size_t len);

Returns: string{str, len}.

u16string operator"s(const char16_t* str, size_t len);

Returns: u16string{str, len}.

u32string operator"s(const char32_t* str, size_t len);

Returns: u32string{str, len}.

wstring operator"s(const wchar_t* str, size_t len);

Returns: wstring{str, len}.

[Note: The same suffix s is used for chrono::duration literals denoting seconds but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — end note]

24.4 String view classes

The class template basic_string_view describes an object that can refer to a constant contiguous sequence of char-like (24.1) 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]

The complexity of basic_string_view member functions is \(O(1)\) unless otherwise specified.
24.4.1 Header <string_view> synopsis

namespace std {
    // 24.4.2, class template basic_string_view
    template<class charT, class traits = char_traits<charT>>
    class basic_string_view;

    // 24.4.3, 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 24.4.3, sufficient additional overloads of comparison functions

    // 24.4.4, 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 u16string_view = basic_string_view<char16_t>;
    using u32string_view = basic_string_view<char32_t>;
    using wstring_view = basic_string_view<wchar_t>;

    // 24.4.5, hash support
    template<class T> struct hash;
    template<> struct hash<string_view>;
    template<> struct hash<u16string_view>;
    template<> struct hash<u32string_view>;
    template<> struct hash<wstring_view>;

    inline namespace literals {
        inline namespace string_view_literals {
            // 24.4.6, suffix for basic_string_literal
            constexpr string_view operator"sv(const char* 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 23.2.2 and 27.7 are available when <string_view> is included.

24.4.2 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 24.4.2.2
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);

// 24.4.2.1, construction and assignment
constexpr basic_string_view() noexcept;
constexpr basic_string_view(const basic_string_view&) noexcept = default;
constexpr basic_string_view(const charT* str);
constexpr basic_string_view(const charT* str, size_type len);

// 24.4.2.2, 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;

// 24.4.2.3, 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;

// 24.4.2.4, 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;

// 24.4.2.5, modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_string_view& s) noexcept;

// 24.4.2.6, string operations
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;

// 24.4.2.5.3, modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_string_view& s) noexcept;

// 24.4.2.6, string operations
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;

232) Because basic_string_view refers to a constant sequence, iterator and const_iterator are the same type.
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;

constexpr size_type find(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find(charT c, size_type pos = 0) const noexcept;
constexpr size_type find(const charT* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type rfind(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind(const charT* s, size_type pos = npos) const;

constexpr size_type find_first_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(charT c, size_type pos = 0) const noexcept;
constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(charT c, size_type pos = npos) const noexcept;
constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;
constexpr size_type find_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
constexpr size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_first_not_of(const charT* s, size_type pos = 0) const;
constexpr size_type find_last_not_of(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
constexpr size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;

private:
    const_pointer data_; // exposition only
    size_type size_;     // exposition only
};

In every specialization basic_string_view<charT, traits>, the type traits shall satisfy the character traits requirements (24.2), and the type traits::char_type shall name the same type as charT.

24.4.2.1 Construction and assignment
[string.view.cons]

constexpr basic_string_view() noexcept;

Effects: Constructs an empty basic_string_view.

Postconditions: size_ == 0 and data_ == nullptr.

constexpr basic_string_view(const charT* str);

Requires: [str, str + traits::length(str)) is a valid range.

Effects: Constructs a basic_string_view, with the postconditions in Table 64.

Table 64 — basic_string_view(const charT*) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_</td>
<td>str</td>
</tr>
<tr>
<td>size_</td>
<td>traits::length(str)</td>
</tr>
</tbody>
</table>

Complexity: \(O(\text{traits::length(str)})\).
constexpr basic_string_view(const charT* str, size_type len);

6  Requires: \([str, str + len)\) is a valid range.

7  Effects: Constructs a basic_string_view, with the postconditions in Table 65.

Table 65 — basic_string_view(const charT*, size_type) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_</td>
<td>str</td>
</tr>
<tr>
<td>size_</td>
<td>len</td>
</tr>
</tbody>
</table>

24.4.2.2  Iterator support

using const_iterator = implementation-defined;

1  A type that meets the requirements of a constant random access iterator (27.2.7) and of a contiguous iterator (27.2.1) 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 (26.2) 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

    — if !empty(), &*begin() == data_,
    — 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()).

24.4.2.3  Capacity

        constexpr size_type size() const noexcept;

1  Returns: size_.

        constexpr size_type length() const noexcept;

2  Returns: size_.

        constexpr size_type max_size() const noexcept;

3  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;

4  Returns: size_ == 0.

24.4.2.4  Element access

        constexpr const_reference operator[](size_type pos) const;

1  Requires: 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;

Requires: !empty().

Returns: data_[0].

Throws: Nothing.

constexpr const_reference back() const;

Requires: !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]

24.4.2.5 Modifiers

constexpr void remove_prefix(size_type n);

Requires: n <= size().

Effects: Equivalent to: data_ += n; size_ -= n;

constexpr void remove_suffix(size_type n);

Requires: n <= size().

Effects: Equivalent to: size_ -= n;

constexpr void swap(basic_string_view& s) noexcept;

Effects: Exchanges the values of *this and s.

24.4.2.6 String operations

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().

Requires: [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:** \(\Theta(rlen)\).

**Returns:** The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 66.

<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 compare(0, npos, x) == 0;

constexpr bool starts_with(charT x) const noexcept;

**Effects:** Equivalent to: return starts_with(basic_string_view(&x, 1));

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:

\[
\begin{align*}
\text{return } \text{size()} \geq x.\text{size()} & \& \text{compare(size() - x.size(), npos, x)} == 0; \\
\text{constexpr bool ends_with(charT x) const noexcept;}
\end{align*}
\]

**Effects:** Equivalent to: return ends_with(basic_string_view(&x, 1));

constexpr bool ends_with(const charT* x) const;

**Effects:** Equivalent to: return ends_with(basic_string_view(x));

24.4.2.7 Searching

This subclause specifies the basic_string_view member functions named find, rfind, find_first_of, find_last_of, find_first_not_of, and find_last_not_of.

Member functions in this subclause have complexity \(\Theta(\text{size()} \times \text{str.size()})\) at worst, although implementations should do better.

§ 24.4.2.7
Each member function of the form

```cpp
constexpr return-type F(const charT* s, size_type pos);
```
is equivalent to return

```cpp
F(basic_string_view(s), pos);
```

Each member function of the form

```cpp
constexpr return-type F(const charT* s, size_type pos, size_type n);
```
is equivalent to return

```cpp
F(basic_string_view(s, n), pos);
```

Each member function of the form

```cpp
constexpr return-type F(charT c, size_type pos);
```
is equivalent to return

```cpp
F(basic_string_view(&c, 1), pos);
```

```cpp
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:

1. `pos <= xpos`
2. `xpos + str.size() <= size`
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`.

```cpp
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:

1. `xpos <= pos`
2. `xpos + str.size() <= size`
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`.

```cpp
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:

1. `pos <= xpos`
2. `xpos < size`
3. `traits::eq(at(xpos), str.at(I))` for some element `I` of the string referenced by `str`.

**Effects:** Determines `xpos`.

**Returns:** `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```cpp
constexpr size_type find_last_of(basic_string_view str, size_type pos = npos) const noexcept;
```

Let `xpos` be the highest position, if possible, such that the following conditions hold:

1. `xpos <= pos`
2. `xpos < size`
3. `traits::eq(at(xpos), str.at(I))` for some element `I` of the string referenced by `str`.

**Effects:** Determines `xpos`.

**Returns:** `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

```cpp
constexpr size_type find_first_not_of(basic_string_view str, size_type pos = 0) const noexcept;
```

Let `xpos` be the lowest position, if possible, such that the following conditions hold:

1. `pos <= xpos`
2. `xpos < size`
3. `traits::eq(at(xpos), str.at(I))` for no element `I` of the string referenced by `str`.
Effects: Determines \( \text{xpos} \).

Returns: \( \text{xpos} \) if the function can determine such a value for \( \text{xpos} \). Otherwise, returns \( \text{npos} \).

```cpp
constexpr size_type find_last_not_of(basic_string_view str, size_type pos = npos) const noexcept;
```

Let \( \text{xpos} \) be the highest position, if possible, such that the following conditions hold:

1. \( \text{xpos} \leq \text{pos} \)
2. \( \text{xpos} < \text{size}() \)
3. \( \text{traits::eq(at(xpos), str.at(I))} \) for no element \( I \) of the string referenced by \( \text{str} \).

Effects: Determines \( \text{xpos} \).

Returns: \( \text{xpos} \) if the function can determine such a value for \( \text{xpos} \). Otherwise, returns \( \text{npos} \).

### 24.4.3 Non-member comparison functions

Let \( S \) be \( \text{basic_string_view<charT, traits>} \), and \( sv \) be an instance of \( S \). Implementations shall provide sufficient additional overloads marked \( \text{constexpr} \) and \( \text{noexcept} \) so that an object \( t \) with an implicit conversion to \( S \) can be compared according to Table 67.

<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 \( \text{operator==} \) would be:

```cpp
template<class T> using __identity = decay_t<T>;
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, __identity<basic_string_view<charT, traits>> rhs) noexcept {
    return lhs.compare(rhs) == 0;
  }
template<class charT, class traits>
  constexpr bool operator==(__identity<basic_string_view<charT, traits>> lhs, basic_string_view<charT, traits> rhs) noexcept {
    return lhs.compare(rhs) == 0;
  }
```

Returns: \( \text{lhs.compare(rhs)} == 0 \).
template<class charT, class traits>
constexpr bool operator!=(basic_string_view<charT, traits> lhs,
    basic_string_view<charT, traits> rhs) noexcept;

Returns: lhs.compare(rhs) != 0.

template<class charT, class traits>
constexpr bool operator<(basic_string_view<charT, traits> lhs,
    basic_string_view<charT, traits> rhs) noexcept;

Returns: lhs.compare(rhs) < 0.

template<class charT, class traits>
constexpr bool operator>(basic_string_view<charT, traits> lhs,
    basic_string_view<charT, traits> rhs) noexcept;

Returns: lhs.compare(rhs) > 0.

template<class charT, class traits>
constexpr bool operator<=(basic_string_view<charT, traits> lhs,
    basic_string_view<charT, traits> rhs) noexcept;

Returns: lhs.compare(rhs) <= 0.

template<class charT, class traits>
constexpr bool operator>=(basic_string_view<charT, traits> lhs,
    basic_string_view<charT, traits> rhs) noexcept;

Returns: lhs.compare(rhs) >= 0.

24.4.4 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 (30.7.5.2.1) 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 30.7.5.2.1. 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

24.4.5 Hash support [string.view.hash]

template<> struct hash<string_view>;
template<> struct hash<u16string_view>;
template<> struct hash<u32string_view>;
template<> struct hash<wstring_view>;

The specialization is enabled (23.14.15). [Note: The hash value of a string view object is equal to the
hash value of the corresponding string object (24.3.5). —end note]

24.4.6 Suffix for basic_string_view literals [string.view.literals]

constexpr string_view operator"sv(const char* str, size_t len) noexcept;

Returns: string_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).
24.5 Null-terminated sequence utilities

24.5.1 Header <ctype> synopsis

```cpp
namespace std {
    int isalnum(int c);
    int isalpha(int c);
    int isblank(int c);
    int iscntrl(int c);
    int isdigit(int c);
    int isgraph(int c);
    int islower(int c);
    int isprint(int c);
    int ispunct(int c);
    int isspace(int c);
    int isupper(int c);
    int isxdigit(int c);
    int tolower(int c);
    int toupper(int c);
}
```

1 The contents and meaning of the header `<ctype>` are the same as the C standard library header `<ctype.h>`.

See also: ISO C 7.4

24.5.2 Header <cwctype> synopsis

```cpp
namespace std {
    using wint_t = see below;
    using wctrans_t = see below;
    using wctype_t = see below;

    int iswalnum(wint_t wc);
    int iswalpha(wint_t wc);
    int iswblank(wint_t wc);
    int iswcntrl(wint_t wc);
    int iswdigit(wint_t wc);
    int iswgraph(wint_t wc);
    int iswlower(wint_t wc);
    int iswprint(wint_t wc);
    int iswpunct(wint_t wc);
    int iswspace(wint_t wc);
    int iswupper(wint_t wc);
    int iswxdigit(wint_t wc);
    int iswctype(wint_t wc, wctype_t desc);
    wctype_t wctype(const char* property);
    wint_t towlower(wint_t wc);
    wint_t towupper(wint_t wc);
    wint_t towctrans(wint_t wc, wctrans_t desc);
    wctrans_t wctrans(const char* property);
}
```

#define WEOF see below

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

24.5.3 Header <cstring> synopsis

```cpp
namespace std {
    using size_t = see 21.2.4;

    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);
}
```
The contents and meaning of the header `<cstring>` are the same as the C standard library header `<string.h>`. The functions `strerror` and `strtok` are not required to avoid data races (20.5.5.9). The functions `memcpy` and `memmove` are signal-safe (21.11.4). [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 (20.2).—end note]

See also: ISO C 7.24

### 24.5.4 Header `<cwchar>` synopsis

```c
namespace std {
    using size_t = see 21.2.4;
    using mbstate_t = see below;
    using wint_t = see below;
    using tm;

    int fputwc(FILE* stream, wchar_t c);
    wchar_t* fgetws(wchar_t* s, size_t n, FILE* stream);
}
```
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`.

1

[Note: The functions `wcschr`, `wcsbrk`, `wcsrchr`, `wcsstr`, and `wmemchr` have different signatures in this document, but they have the same behavior as in the C standard library (20.2). —end note]

See also: ISO C 7.29
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 does not declare types char16_t nor char32_t.

See also: ISO C 7.28

24.5.6 Multibyte / wide string and character conversion functions [c.mb.wcs]

1 [Note: The headers <cstdlib> (21.2.2) and <cwchar> (24.5.4) declare the functions described in this subclause. — end note]

int mbsinit(const mbstate_t* ps);
int mblen(const char* s, size_t n);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n);
size_t wcstombs(char* s, const wchar_t* pwcs, size_t n);

2 Effects: These functions have the semantics specified in the C standard library.

See also: ISO C 7.22.7, 7.22.8, 7.29.6.2.1

int mbtowc(wchar_t* pwc, const char* s, size_t n);
int wctomb(char* s, wchar_t wchar);

3 Effects: These functions have the semantics specified in the C standard library.

Remarks: Calls to these functions may introduce a data race (20.5.5.9) with other calls to the same function.

See also: ISO C 7.22.7

size_t mbtowc(const char* s, size_t n, mbstate_t* ps);
size_t mbstowcs(wchar_t* pwcs, const char* s, size_t n, mbstate_t* ps);
size_t wcstombs(char* s, wchar_t wc, mbstate_t* ps);
size_t mbstowcs(wchar_t* dst, const char** src, size_t len, mbstate_t* ps);
size_t wcstombs(char* dst, const wchar_t** src, size_t len, mbstate_t* ps);

5 Effects: These functions have the semantics specified in the C standard library.

6 Remarks: Calling these functions with an mbstate_t* argument that is a null pointer value may introduce a data race (20.5.5.9) 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
25 Localization library

25.1 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 68.

Table 68 — Localization library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.3</td>
<td>&lt;locale&gt;</td>
</tr>
<tr>
<td>25.4</td>
<td>Standard locale Categories</td>
</tr>
<tr>
<td>25.5</td>
<td>&lt;locale&gt;</td>
</tr>
</tbody>
</table>

25.2 Header <locale> synopsis

```cpp
namespace std {
    // 25.3.1, locale
    class locale;
    template<class Facet> const Facet& use_facet(const locale&);
    template<class Facet> bool has_facet(const locale&) noexcept;
    // 25.3.3, 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);
    // 25.4.1, 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;
    // 25.4.2, numeric
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class num_get;
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class num_put;
    template<class charT>
    class numprint;
```
template<class charT>
class numpunct_byname;

// 25.4.4, collation
template<class charT> class collate;
template<class charT> class collate_byname;

// 25.4.5, 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;

// 25.4.6, 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;

// 25.4.7, 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.233

25.3 Locales

25.3.1 Class locale

namespace std {
class locale {
public:
    // types
    class facet;
    class id;
    using category = int;
    static const category // values assigned here are for exposition only
        none = 0,
        collate = 0x010, ctype = 0x020,
        monetary = 0x040, numeric = 0x080,
        time = 0x100, messages = 0x200,
        all = collate = ctype = monetary = numeric = time = messages;

    // construct/copy/destroy
    locale() noexcept;
    locale(const locale& other) noexcept;
    explicit locale(const char* std_name);
    explicit locale(const string& std_name);
    locale(const locale& other, const char* std_name, category);
    locale(const locale& other, const string& std_name, category);
    template<class Facet> locale(const locale& other, Facet* f);

233) In this subclause, the type name struct tm is an incomplete type that is defined in <ctime>.
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;

// static locale objects
static locale global(const locale&);
static const locale& classic();
};

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.

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]

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 (25.4.8).

[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 (30.7.4.2)).

—end note]

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.

234) Note that in the call to `put` the stream is implicitly converted to an `ostreambuf_iterator<charT, traits>`. 

§ 25.3.1 710
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 (20.5.5.9).

25.3.1.1 locale types

25.3.1.1.1 Type `locale::category`

```cpp
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

\[(\text{collate} | \text{ctype} | \text{monetary} | \text{numeric} | \text{time} | \text{messages} | \text{all}) == \text{all}\]

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 69.

<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, char, mbstate_t&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>codecvt&lt;char32_t, char, 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 69, `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 70.

The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(1)` only for facet `F` of types `numpunct<charT>` and `ctype<charT>`, and for locale `l` the value obtained by calling `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 requirements of an Input Iterator or an
Table 70 — 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, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;char32_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td></td>
<td>codecvt_byname&lt;wchar_t, char, mbstate_t&gt;</td>
</tr>
<tr>
<td>monetary</td>
<td>moneypunct_byname&lt;char, International&gt;</td>
</tr>
<tr>
<td></td>
<td>moneypunct_byname&lt;wchar_t, International&gt;</td>
</tr>
<tr>
<td></td>
<td>money_get&lt;C, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>money_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>numeric</td>
<td>numpunct_byname&lt;char&gt;, numpunct_byname&lt;wchar_t&gt;</td>
</tr>
<tr>
<td></td>
<td>num_get&lt;C, InputIterator&gt;, num_put&lt;C, OutputIterator&gt;</td>
</tr>
<tr>
<td>time</td>
<td>time_get&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;char, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_get_byname&lt;wchar_t, InputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;char, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td></td>
<td>time_put_byname&lt;wchar_t, OutputIterator&gt;</td>
</tr>
<tr>
<td>messages</td>
<td>messages_byname&lt;char&gt;, messages_byname&lt;wchar_t&gt;</td>
</tr>
</tbody>
</table>

Output Iterator, respectively (27.2). 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.

25.3.1.1.2 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.

235) 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.

§ 25.3.1.1.2
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.

25.3.1.1.3 Class `locale::id`

```cpp
namespace std {
    class locale::id {
    public:
        id();
        void operator=(const id&) = delete;
        id(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]

25.3.1.2 `locale` 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()`.

[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;
```

3 `Effects: Constructs a locale which is a copy of `other`.

```cpp
explicit locale(const char* std_name);
```

4 `Effects: Constructs a locale using standard C locale names, e.g., "POSIX". The resulting locale implements semantics defined to be associated with that name.

5 `Throws: runtime_error` if the argument is not valid, or is null.

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 `Throws: runtime_error` if the argument is not valid, or is null.

10 `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);
```

11 `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.

~locale();

A non-virtual destructor that throws no exceptions.

25.3.1.3 locale 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 ")".

25.3.1.4 locale 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 28) 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 (28.7.1, 26.3.11):

std::sort(v.begin(), v.end(), loc);

— end example]
25.3.1.5 locale static members

static locale global(const locale& loc);
1
Sets the global locale to its argument.
2
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 25.5 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.

25.3.2 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 25.3.1.1.2.
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.

25.3.3 Convenience interfaces

25.3.3.1 Character classification

template<class charT> bool isspace(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 (25.4.1).\textsuperscript{236}

25.3.3.2 Conversions

25.3.3.2.1 Character conversions

template<class charT> charT toupper(charT c, const locale& loc);
1
Returns: use_facet<ctype<charT>>(loc).toupper(c).

\textsuperscript{236} When used in a loop, it is faster to cache the ctype<> facet and use it directly, or use the vector form of ctype<>::is.
template<class charT> char tolower(charT c, const locale& loc);

2

Returns: use_facet<ctype<charT>>(loc).tolower(c).

25.4 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 (30.5.3). 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.

2 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.

3 Within this clause it is unspecified whether one virtual function calls another virtual function.

25.4.1 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;
    }
}

1 The type mask is a bitmask type (20.4.2.1.4).

25.4.1.1 Class template ctype

namespace std {
    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* scan_not(charT c)

                    const charT* tolower(charT c) const;

                    const char* widen(const char* low, const char* high, char* 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 69 (25.3.1.1.1), namely ctype<char> and ctype<wchar_t>, implement character classing appropriate to the implementation's native character set.

25.4.1.1.1 ctype members

bool is(mask m, charT c) 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) const;
const charT* toupper(charT* low, const charT* high) const;
const charT* tolower(charT) const;
const charT* tolower(charT* low, const charT* high) const;
const charT* widen(const char* low, const char* high, charT* dest) const;
const charT* widen(const char* low, const char* high, charT* dest) const;
char narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT* high, char dfault, char* dest) const;

25.4.1.1.2 ctype virtual functions

bool do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high, mask* vec) const;

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 \((M \& m) \neq 0\); i.e., \texttt{true} if the character has the characteristics specified. The second form returns \texttt{high}.

\begin{verbatim}
const charT* 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 \(m\).

Returns: The smallest pointer \(p\) in the range \([\texttt{low}, \texttt{high})\) such that \(\texttt{is}(m, \ast p)\) would return \texttt{true}; otherwise, returns \texttt{high}.

\begin{verbatim}
const charT* 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 \(m\).

Returns: The smallest pointer \(p\), if any, in the range \([\texttt{low}, \texttt{high})\) such that \(\texttt{is}(m, \ast p)\) would return \texttt{false}; otherwise, returns \texttt{high}.

\begin{verbatim}
charT do_toupper(charT c) const;
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 \(\ast p\) in the range \([\texttt{low}, \texttt{high})\) 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 \texttt{high}.

\begin{verbatim}
charT do_tolower(charT c) const;
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 \(\ast p\) in the range \([\texttt{low}, \texttt{high})\) 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 \texttt{high}.

\begin{verbatim}
charT do_widen(char c) const;
const charT* do_widen(const char* low, const char* high, charT* dest) const;
\end{verbatim}

Effects: Applies the simplest reasonable transformation from a \texttt{char} value or sequence of \texttt{char} values to the corresponding \texttt{charT} value or values.\(^{237}\) The only characters for which unique transformations are required are those in the basic source character set (5.3).

For any named \texttt{ctype} category with a \texttt{ctype <charT>} facet \(\texttt{ctc}\) and valid \texttt{ctype_base::mask} value \(M\), \((\texttt{ctc.is}(M, c) \mid \mid \texttt{!is}(M, \texttt{do_widen}(c))\)) is \texttt{true}.\(^{238}\)

The second form transforms each character \(\ast p\) in the range \([\texttt{low}, \texttt{high})\), placing the result in \(\texttt{dest}[p - \texttt{low}]\).

Returns: The first form returns the transformed value. The second form returns \texttt{high}.

\begin{verbatim}
char do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high, char dfault, char* dest) const;
\end{verbatim}

Effects: Applies the simplest reasonable transformation from a \texttt{charT} value or sequence of \texttt{charT} values to the corresponding \texttt{char} value or values.

For any character \(c\) in the basic source character set (5.3) the transformation is such that \(\texttt{do_widen(\texttt{do_narrow}(c, 0)) == c}\).

For any named \texttt{ctype} category with a \texttt{ctype<char>} facet \(\texttt{ctc}\) however, and \texttt{ctype_base::mask} value \(M\), \((\texttt{is}(M, c) \mid \mid \texttt{!ctc.is}(M, \texttt{do_narrow}(c, \texttt{dfault}))\)) is \texttt{true} (unless \texttt{do_narrow} returns \texttt{default}). In addition, for any digit character \(c\), the expression \((\texttt{do_narrow}(c, \texttt{default}) - '0')\) evaluates to the digit value of the character. The second form transforms each character \(\ast p\) in the range \([\texttt{low}, \texttt{high})\), placing the result (or \texttt{default} if no simple transformation is readily available) in \(\texttt{dest}[p - \texttt{low}]\).

\(^{237}\) The char argument of \texttt{do_widen} is intended to accept values derived from character literals for conversion to the locale’s encoding.

\(^{238}\) 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 \texttt{default} if no mapping is readily available. The second form returns \texttt{high}.

25.4.1.2 Class template \texttt{ctype\_byname}

namespace std {
    template<class charT>
    class ctype\_byname : public ctype<charT> {
        public:
            using mask = typename ctype<charT>::mask;
            explicit ctype\_byname(const char*, size\_t refs = 0);
            explicit ctype\_byname(const string&, size\_t refs = 0);

            protected:
                ~ctype\_byname();
            }
    }

25.4.1.3 \texttt{ctype} specializations

namespace std {
    template<>
    class ctype<char> : public locale::facet, public ctype\_base {
        public:
            using char\_type = char;
            explicit ctype(const mask* tab = nullptr, bool del = false, size\_t refs = 0);

            bool is(mask m, char c) const;
            const char* is(const char* low, const char* high, mask* vec) const;
            const char* scan\_is (mask m, const char* low, const char* high) const;
            const char* scan\_not(mask m, const char* low, const char* high) const;

            char toupper(char c) const;
            const char* toupper(char* low, const char* high) const;
            char tolower(char c) const;
            const char* tolower(char* low, const char* high) const;

            char widen(char c) const;
            const char* widen(const char* low, const char* high, char* to) const;
            char narrow(char c, char default) const;
            const char* narrow(const char* low, const char* high, char default, char* to) const;

            static locale::id id;
            static const size\_t table\_size = implementation\_defined;

            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 default) const;
                virtual const char* do\_narrow(const char* low, const char* high,
                                             char default, char* to) const;
            }
    }

§ 25.4.1.3
A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented inline.\(^{239}\) The implementation-defined value of member `table_size` is at least 256.

### 25.4.1.3.1 `ctype<char>` destructor

```
~ctype();
```

**Effects:** If the constructor’s first argument was nonzero, and its second argument was `true`, does `delete [] table();`.

### 25.4.1.3.2 `ctype<char>` members

In the following member descriptions, for `unsigned char` values `v` where `v >= table_size`, `table()[v]` is assumed to have an implementation-specific value (possibly different for each such value `v`) without performing the array lookup.

```
explicit ctype(const mask* tbl = nullptr, bool del = false, size_t refs = 0);
```

**Requires:** `tbl` either 0 or an array of at least `table_size` elements.

**Effects:** Passes its `refs` argument to its base class constructor.

```
bool is(mask m, char c) const;
const char* is(const char* low, const char* high, mask* vec) const;
```

**Effects:** The second form, for all `*p` in the range `[low, high)`, assigns into `vec[p - low]` the value `table()[(unsigned char)*p]`.

**Returns:** The first form returns `table()[(unsigned char)c] & m`; the second form returns `high`.

```
const char* scan_is(mask m, const char* low, const char* high) const;
```

**Returns:** The smallest `p` in the range `[low, high)` such that 
`table()[(unsigned char) *p] & m` is `true`.

```
const char* scan_not(mask m, const char* low, const char* high) const;
```

**Returns:** The smallest `p` in the range `[low, high)` such that 
`table()[(unsigned char) *p] & m` is `false`.

```
toupper(char c) const;
toupper(char* low, const char* high) const;
```

**Returns:** `do_toupper(c)` or `do_toupper(low, high)`, respectively.

```
tolower(char c) const;
tolower(char* low, const char* high) const;
```

**Returns:** `do_tolower(c)` or `do_tolower(low, high)`, respectively.

```
widen(char c) const;
widen(const char* low, const char* high, char* to) const;
```

**Returns:** `do_widen(c)` or `do_widen(low, high, to)`, respectively.

```
narrow(char c, char dfault) const;
narrow(const char* low, const char* high, char dfault, char* to) const;
```

**Returns:** `do_narrow(c, dfault)` or `do_narrow(low, high, dfault, to)`, respectively.

```
const mask* table() const noexcept;
```

**Returns:** The first constructor argument, if it was nonzero, otherwise `classic_table()`.

---

\(^{239}\) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`. 
25.4.1.3.3 ctype<char> static members

static const mask* classic_table() noexcept;

Returns: A pointer to the initial element of an array of size table_size which represents the classifications of characters in the "C" locale.

25.4.1.3.4 ctype<char> virtual functions

These functions are described identically as those members of the same name in the ctype class template (25.4.1.1.1).

25.4.1.4 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& state, const externT* from, const externT* end, size_t max) const;
            int max_length() const noexcept;

            static locale::id id;

    protected:
        ~codecvt();

        virtual result do_out(
            stateT& state,
            const internT* from, const internT* from_end, const internT*& from_next,
            externT* to, externT* to_end, externT*& to_next) const;

§ 25.4.1.4
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.

The `stateT` argument selects the pair of character encodings being mapped between.

The specializations required in Table 69 (25.3.1.1.1) 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, 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. `codecvt<wchar_t, char, mbstate_t>` converts between the native character sets for narrow 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 user-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.

### 25.4.1.4.1 codecvt members

```
1 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;

2 virtual result do_unshift(
    stateT& state,
    externT* to, externT* to_end, externT*& to_next) const;

3 virtual int do_encoding() const noexcept;

4 virtual bool do_always_noconv() const noexcept;

5 virtual int do_length(stateT, const externT* from, const externT* end, size_t max) const;

6 virtual int do_max_length() const noexcept;
```

§ 25.4.1.4.1
25.4.1.4.2 codecvt 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 (30.9) shall have the property that if
do_out(state, from, from_end, 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, next, to, to_end, to_next)
would return ok, where to != to_end, then

do_in(state, from, from_end, next, to, to + 1, to_next)
shall also return ok. [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 71.

<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.

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 72.

<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>::do_length() 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.

### 25.4.1.5 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();

};

241) Typically these will be characters to return the state to stateT().
242) 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.
25.4.2 The numeric category

The classes `num_get<>` and `num_put<>` handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.\(^{243}\)

All specifications of member functions for `num_put` and `num_get` in the subclauses of 25.4.2 only apply to the specializations required in Tables 69 and 70 (25.3.1.1.1), 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 (25.4), 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.

Extractor and inserter members of the standard iostreams use `num_get<>` and `num_put<>` member functions for formatting and parsing numeric values (30.7.4.2, 30.7.5.2.1).

25.4.2.1 Class template `num_get`  

```cpp
namespace std {
  template<class charT, class InputIterator = istreambuf_iterator<charT>>
  class num_get : public locale::facet {
  public:
    using char_type = charT;
    using iter_type = InputIterator;

    explicit num_get(size_t refs = 0);

    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, bool& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, unsigned short& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, unsigned int& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, unsigned long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, unsigned long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, float& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, long double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
      ios_base::iostate& err, void*& v) const;

    static locale::id id;

  protected:
    //num_get();
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, bool& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, long long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, unsigned short& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, unsigned int& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, unsigned long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, unsigned long long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, float& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, long double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
      ios_base::iostate& err, void*& v) const;
  }
}
```

\(^{243}\) 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;
};

The facet num_get is used to parse numeric values from an input sequence such as an istream.

25.4.2.1.1 num_get members

iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
    ios_base::iostate& err, void*& val) const;

Returns: do_get(in, end, str, err, val).

25.4.2.1.2 num_get 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<
    charT>>(loc), and use_facet<numpunct<charT>>(loc), where loc is str.getloc().

The details of this operation occur in three stages

(2.1) — Stage 1: Determine a conversion specifier

(2.2) — Stage 2: Extract characters from in and determine a corresponding char value for the format
    expected by the conversion specification determined in stage 1.

(2.3) — Stage 3: Store results

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 73. The table is ordered. That is, the first line whose condition is true applies.

<table>
<thead>
<tr>
<th>State</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>basefield == oct</td>
<td>%o</td>
</tr>
<tr>
<td>basefield == hex</td>
<td>%X</td>
</tr>
<tr>
<td>basefield == 0</td>
<td>%i</td>
</tr>
<tr>
<td>signed integral type</td>
<td>%d</td>
</tr>
<tr>
<td>unsigned integral type</td>
<td>%u</td>
</tr>
</tbody>
</table>

For conversions to a floating 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 74.

<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 ',' has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if ',' has already been accumulated, the character is discarded and Stage 2 terminates. If it is not discarded, then a check is made to determine if \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{itemize}
  \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{itemize}

The numeric value to be stored can be one of:

\begin{itemize}
  \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{itemize}

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<
\texttt{numpunct<charT>}>}(\texttt{loc}).\texttt{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} == \texttt{end} then \texttt{err} |= \texttt{ios_base::eofbit} is performed.

\begin{verbatim}
iter_type do_get(iter_type in, iter_type end, ios_base& str, 
  ios_base::iostate& err, bool& val) const;
\end{verbatim}

\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<charT>>(str.getloc())}. Successive characters in the range \texttt{[in, end]} (see 26.2.3) 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} == \texttt{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} == \texttt{end}). [Example: For targets \texttt{true: a} and false: "abb", the input sequence "a" yields \texttt{val} == \texttt{true} and \texttt{err} == \texttt{str.eofbit}; the input sequence "abc" yields \texttt{err} == \texttt{str.failbit}, with \texttt{in} ending at the 'c' element. For targets \texttt{true: l} and false: "0", the input sequence "1" yields \texttt{val} == \texttt{true} and \texttt{err} == \texttt{str.goodbit}. For empty targets (""), any input sequence yields \texttt{err} == \texttt{str.failbit}. — end example]

\textbf{Returns:} \texttt{in}.
25.4.2.2 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;

            static locale::id id;

        protected:
            ~num_put();
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, bool val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, long val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, long long val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, double val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, long double val) const;
            virtual iter_type do_put(iter_type out, ios_base& str, char_type fill, const void* val) const;
    }
}

The facet num_put is used to format numeric values to a character sequence such as an ostream.

25.4.2.2.1 num_put members

    iter_type put(iter_type out, ios_base& str, char_type fill, bool val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, long long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, double val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, long double val) const;
    iter_type put(iter_type out, ios_base& str, char_type fill, const void* val) const;

1 Returns: do_put(out, str, fill, val).

25.4.2.2.2 num_put virtual functions

    iter_type do_put(iter_type out, ios_base& str, char_type fill, long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, long long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, unsigned long long val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, double val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, long double val) const;
    iter_type do_put(iter_type out, ios_base& str, char_type fill, const void* val) const;

1 Effects: Writes characters to the sequence out, formatting val as desired. In the following description, a local variable initialized with:

    locale loc = str.getloc();

2 The details of this operation occur in several stages:

   — Stage 1: Determine a printf conversion specifier spec and determine the characters that would be printed by printf (30.12) 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>>(str.getloc())

— Stage 3: Determine where padding is required.

— Stage 4: Insert the sequence into the out.

Detailed descriptions of each stage follow.

Returns: out.

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe
this determination use the following local variables

fmtflags flags = str.flags();
fmtflags basefield = (flags & (ios_base::basefield));
fmtflags uppercase = (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos = (flags & (ios_base::showpos));
fmtflags showbase = (flags & (ios_base::showbase));
fmtflags showpoint = (flags & (ios_base::showpoint));

All tables used in describing stage 1 are ordered. That is, the first line whose condition is true
applies. A line without a condition is the default behavior when none of the earlier lines apply.
For conversion from an integral type other than a character type, the function determines the
integral conversion specifier as indicated in Table 75.

<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; uppercases</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 76.

<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; !uppercases</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; !uppercases</td>
</tr>
<tr>
<td>floatfield == (ios_base::fixed</td>
<td>ios_base::scientific)</td>
</tr>
<tr>
<td>!uppercases</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 77.

The conversion specifier has the following optional additional qualifiers prepended as indicated in
Table 78.

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 77 — 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 78 — Numeric conversions

<table>
<thead>
<tr>
<th>Type(s)</th>
<th>State</th>
<th>_stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>an integral type</td>
<td>showpos</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>showbase</td>
<td>#</td>
</tr>
<tr>
<td>a floating-point type</td>
<td>showpos</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>showpoint</td>
<td>#</td>
</tr>
</tbody>
</table>

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.getloc());
```

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 25.4.3.1.2

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 padding244 is determined according to Table 79.

Table 79 — Fill padding

<table>
<thead>
<tr>
<th>State</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjustfield == ios_base::left</td>
<td>pad after</td>
</tr>
<tr>
<td>adjustfield == ios_base::right</td>
<td>pad before</td>
</tr>
<tr>
<td>adjustfield == internal and a sign occurs in the representation</td>
<td>pad after the sign</td>
</tr>
<tr>
<td>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
```

6 Returns: If (str.flags() & ios_base::boolalpha) == 0 returns do_put(out, str, fill, (int)val), otherwise obtains a string s as if by

244) 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.

25.4.3 The numeric punctuation facet

25.4.3.1 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; // for bool
        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 69 (25.3.1.1.1), 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]

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.

25.4.3.1.1 numpunct members

char_type decimal_point() const;

    Returns: do_decimal_point().

1

char_type thousands_sep() const;

    Returns: do_thousands_sep().

2
string grouping() const;

Returns: do_grouping().

string_type truename() const;
string_type falsename() const;

Returns: do_truename() or do_falsename(), respectively.

25.4.3.1.2 numpunct virtual functions [facet.numpunct.virtuals]

char_type do_decimal_point() const;

Returns: A character for use as the decimal radix separator. The required specializations return '.' or L'.'.

char_type do_thousands_sep() const;

Returns: A character for use as the digit group separator. The required specializations return ',' or L','.

string do_grouping() const;

Returns: A basic_string<char> vec used as a vector of integer values, in which each element vec[i] represents the number of digits in the group at position i, starting with position 0 as the rightmost group. If vec.size() <= i, the number is the same as group (i - 1); if (i < 0 || vec[i] <= 0 || vec[i] == CHAR_MAX), the size of the digit group is unlimited.

The required specializations return the empty string, indicating no grouping.

string_type do_truename() const;
string_type do_falsename() const;

Returns: A string representing the name of the boolean value true or false, respectively.

In the base class implementation these names are "true" and "false", or L"true" and L"false".

25.4.3.2 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();
    };
}

25.4.4 The collate category [category.collate]

25.4.4.1 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);

    };
}

245) Thus, the string "\003" specifies groups of 3 digits each, and "3" probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of "3."
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 28) and containers operating on strings. The specializations required in Table 69 (25.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (28.7.10).

Each function compares a string of characters `*p` in the range `[low, high)`.

### 25.4.4.1.1 collate members

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>int compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;</code></td>
</tr>
<tr>
<td>2</td>
<td><code>string_type transform(const charT* low, const charT* high) const;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>long hash(const charT* low, const charT* high) const;</code></td>
</tr>
</tbody>
</table>

**Returns:**

- `int compare()` returns:
  - 1 if the first string is greater than the second,
  - -1 if less,
  - zero otherwise. The specializations required in Table 69 (25.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (28.7.10).

- `string_type transform()` returns:
  - A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `do_compare()` on the same two strings.

- `long hash()` returns:
  - An integer value equal to the result of calling `hash()` on any other string for which `do_compare()` returns 0 (equal) when passed the two strings. [Note: The probability that the result equals that for another string which does not compare equal should be very small, approaching `(1.0/narrow_limits<unsigned(long> :: max())`. —end note]

### 25.4.4.1.2 collate virtual functions

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>int do_compare(const charT* low1, const charT* high1, const charT* low2, const charT* high2) const;</code></td>
</tr>
<tr>
<td>2</td>
<td><code>string_type do_transform(const charT* low, const charT* high) const;</code></td>
</tr>
<tr>
<td>3</td>
<td><code>long do_hash (const charT* low, const charT* high) const;</code></td>
</tr>
</tbody>
</table>

**Returns:**

- `int do_compare()` returns:
  - 1 if the first string is greater than the second,
  - -1 if less,
  - zero otherwise. The specializations required in Table 69 (25.3.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (28.7.10).

- `string_type do_transform()` returns:
  - A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `do_compare()` on the same two strings.

- `long do_hash()` returns:
  - An integer value equal to the result of calling `hash()` on any other string for which `do_compare()` returns 0 (equal) when passed the two strings. [Note: The probability that the result equals that for another string which does not compare equal should be very small, approaching `(1.0/narrow_limits<unsigned(long> :: max())`. —end note]

### 25.4.4.2 Class template `collate_byname`

```cpp
namespace std {
    template<class charT>
    class collate_byname : public collate<charT> {
    public:
        using string_type = basic_string<charT>;
    }
}
```

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();
};

25.4.5 The time category
1 Templates time_get<charT, InputIterator> and time_put<charT, OutputIterator> provide date and time formatting and parsing. All specifications of member functions for time_put and time_get in the subclauses of 25.4.5 only apply to the specializations required in Tables 69 and 70 (25.3.1.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in 25.4, and the ctype<> facet, to determine formatting details.

25.4.5.1 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, char format, char modifier = 0) 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;
};
virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
   ios_base::iostate& err, tm* t, char format, char modifier) const;
};

1 time_get is used to parse a character sequence, extracting components of a time or date into a struct tm object. Each get member parses a format as produced by a corresponding format specifier to time_put<>::put. If the sequence being parsed matches the correct format, the corresponding members of the struct tm argument are set to the values used to produce the sequence; otherwise either an error is reported or unspecified values are assigned.\footnote{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.}

2 If the end iterator is reached during parsing by any of the get() member functions, the member sets ios_base::eofbit in err.

25.4.5.1.1 time_get members

Returns: do_date_order().

1 Returns: do_get_time(s, end, str, err, t).

2 Returns: do_get_time(s, end, str, err, t).

3 Returns: do_get_date(s, end, str, err, t).

4 Returns: do_get_weekday(s, end, str, err, t) or do_get_monthname(s, end, str, err, t).

5 Returns: do_get_year(s, end, str, err, t).

6 Returns: do_get(s, end, f, err, t, format, modifier).

7 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
the optional modifier is absent from the conversion specification. If \( \text{err} == \text{ios\_base::goodbit} \) holds after the evaluation of the expression, the function increments \( \text{fmt} \) to point just past the end of the conversion specification and continues looping.

(8.5) — 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 \( s \) until \( s == \text{end} \) || \( !\text{isspace(}*s, \text{f.getloc()}\) is \( \text{true} \), and finally resumes looping.

(8.6) — The next character read from \( s \) matches the element pointed to by \( \text{fmt} \) in a case-insensitive comparison, in which case the function evaluates \( ++\text{fmt}, ++s \) and continues looping. Otherwise, the function evaluates \( \text{err} == \text{ios\_base::failbit} \).

[Note: The function uses the \( \text{ctype<charT>} \) facet installed in \( f \)'s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so. — end note]

9

Returns: \( s \).

25.4.5.1.2 time_get virtual functions [locale.time.get.virtuals]

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.\(^{248}\) Returns \text{no\_order} if the date format specified by ‘\( x \)’ contains other variable components (e.g., Julian day, week number, week day).

iter_type do_get_time(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

2 Effects: Reads characters starting at \( s \) until it has extracted those \text{struct tm} members, and remaining format characters, used by \text{time\_put<>::put} to produce the format specified by “\%H:\%M:\%S”, or until it encounters an error or end of sequence.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

iter_type do_get_date(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

4 Effects: Reads characters starting at \( s \) until it has extracted those \text{struct tm} members and remaining format characters used by \text{time\_put<>::put} to produce one of the following formats, or until it encounters an error. The format depends on the value returned by \text{date\_order()} as shown in Table 80.

<table>
<thead>
<tr>
<th>date_order()</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_order</td>
<td>“%m%d%y”</td>
</tr>
<tr>
<td>dmy</td>
<td>“%d%m%y”</td>
</tr>
<tr>
<td>mdy</td>
<td>“%m%d%y”</td>
</tr>
<tr>
<td>ymd</td>
<td>“%y%m%d”</td>
</tr>
<tr>
<td>ydm</td>
<td>“%y%dm%d”</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.

iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str, ios_base::iostate& err, tm* t) const;

7 Effects: Reads characters starting at \( s \) until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full

\(^{248}\) This function is intended as a convenience only, for common formats, and may return \text{no\_order} in valid locales.

\( \text{§ 25.4.5.1.2} \)
name, it continues reading until it matches the full name or fails. It sets the appropriate struct tm member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid name.

iter_type do_get_year(iter_type s, iter_type end, ios_base& str,
                     ios_base::iostate& err, tm* t) const;

Effects: Reads characters starting at s until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the t->tm_year member accordingly.

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

iter_type do_get(iter_type s, iter_type end, ios_base& f,
                 ios_base::iostate& err, tm* t, char format, char modifier) const;

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 strftime, formed by concatenating '%', the modifier character, when non-NUL, and the format character. When the concatenation fails to yield a complete valid directive the function leaves the object pointed to by t unchanged and evaluates err |= ios_base::failbit. When s == end evaluates to true after reading a character the function evaluates err |= ios_base::eofbit.

For complex conversion directives such as %c, %x, or %X, or directives that involve the optional modifiers E or O, when the function is unable to unambiguously determine some or all struct tm members from the input sequence [s, end], it evaluates err |= ios_base::eofbit. In such cases the values of those struct tm members are unspecified and may be outside their valid range.

Remarks: It is unspecified whether multiple calls to do_get() with the address of the same struct tm object will update the current contents of the object or simply overwrite its members. Portable programs should zero out the object before invoking the function.

Returns: An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given format and modifier.

25.4.5.2 Class template time_get_byname

namespace std {
    template<class charT, class InputIterator = istreambuf_iterator<charT>>
    class time_get_byname : public time_get<charT, InputIterator> {
        public:
            using dateorder = time_base::dateorder;
            using iter_type = InputIterator;

            explicit time_get_byname(const char*, size_t refs = 0);
            explicit time_get_byname(const string&, size_t refs = 0);

            protected:
                ~time_get_byname();
                
    };
}

25.4.5.3 Class template time_put

namespace std {
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class time_put : public locale::facet {
        public:
            using char_type = charT;
            using iter_type = OutputIterator;

            explicit time_put(size_t refs = 0);
        };
}
// the following is implemented in terms of other member functions.
iter_type put(iter_type s, ios_base& f, 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;
};

25.4.5.3.1 time_put 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;

1 Effects: The first form steps through the sequence from pattern to pat_end, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to s immediately, and each format sequence, as it is identified, results in a call to do_put; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character c to a char value as if by ct.narrow(c, 0), where ct is a reference to ctype<charT> obtained from str.getloc(). The first character of each sequence is equal to '%', followed by an optional modifier character mod and a format specifier character spec as defined for the function strftime. If no modifier character is present, mod is zero. For each valid format sequence identified, calls do_put(s, str, fill, t, spec, mod).

2 The second form calls do_put(s, str, fill, t, format, modifier).

3 [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]

4 Returns: An iterator pointing immediately after the last character produced.

25.4.5.3.2 time_put virtual functions

iter_type do_put(iter_type s, ios_base& str, char_type fill, const tm* t,
    char format, char modifier) const;

1 Effects: Formats the contents of the parameter t into characters placed on the output sequence s. Formatting is controlled by the parameters format and modifier, interpreted identically as the format specifiers in the string argument to the standard library function strftime(), except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.

2 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]

25.4.5.4 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;

        249) Although the C programming language defines no modifiers, most vendors do.
        250) Interpretation of the modifier argument is implementation-defined, but should follow POSIX conventions.
        251) 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();
};

## 25.4.6 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 25.4.6 only apply to the specializations required in Tables 69 and 70 (25.3.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in 25.4, and the `moneypunct<>` and `ctype<>` facets, to determine formatting details.

### 25.4.6.1 Class template `money_get`

```cpp
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 s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err, long double& units) const;
      virtual iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& str, ios_base::iostate& err, string_type& digits) const;
    }
  }
}
```

### 25.4.6.1.1 `money_get` members

- `iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, long double& units) const`;
- `iter_type get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, string_type& digits) const`;

**Returns:** `do_get(s, end, intl, f, err, quant)`.

### 25.4.6.1.2 `money_get` virtual functions

- `iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, long double& units) const`;
- `iter_type do_get(iter_type s, iter_type end, bool intl, ios_base& f, ios_base::iostate& err, string_type& digits) const`;

**Effects:** Reads characters from `s` to parse and construct a monetary value according to the format specified by a `moneypunct<charT, Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()` and `str.flags()`.

§ 25.4.6.1.2
If a valid sequence is recognized, does not change err; otherwise, sets err to (err|str.failbit), or (err|str.failbit|str.eofbit) if no more characters are available, and does not change units or digits. Uses the pattern returned by mp.neg_format() to parse all values. The result is returned as an integral value stored in units or as a sequence of digits possibly preceded by a minus sign (as produced by ct.widen(c) where c is '−' or in the range from '0' through '9', inclusive) stored in digits. [Example: The sequence $1,056.23 in a common United States locale would yield, for units, 105623, or, for digits, "105623". — end example] If mp.grouping() indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

Where money_base::space or money_base::none appears as the last element in the format pattern, no white space is consumed. Otherwise, where money_base::space appears in any of the initial elements of the format pattern, at least one white space character is required. Where money_base::none appears in any of the initial elements of the format pattern, white space is allowed but not required. If (str.flags() & str.showbase) is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

If the first character (if any) in the string pos returned by mp.positive_sign() or the string neg returned by mp.negative_sign() is recognized in the position indicated by sign in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [Example: If showbase is off, then for a neg value of "()" and a currency symbol of "L", in "(100 L)" the "L" is consumed; but if neg is "-", the "L" in "-100 L" is not consumed. — end example] If pos or neg is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of pos or neg, and the result is given the corresponding sign. If the first character of pos is equal to the first character of neg, or if both strings are empty, the result is given a positive sign.

Digits in the numeric monetary component are extracted and placed in digits, or into a character buffer buf1 for conversion to produce a value for units, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value units is produced as if by

```c++
for (int i = 0; i < n; ++i)
    buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);
```

where n is the number of characters placed in buf1, buf2 is a character buffer, and the values src and atoms are defined as if by

```c++
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

### § 25.4.6.2 Class template money_put
[locale.money.put]

```c++
namespace std {
    template<class charT, class OutputIterator = ostreambuf_iterator<charT>>
    class money_put : public locale::facet {
        public:
            using char_type = charT;
            using iter_type = OutputIterator;
            using string_type = basic_string<charT>;

            explicit money_put(size_t refs = 0);

            iter_type put(iter_type s, bool intl, ios_base& f,
                char_type fill, long double units) const;
            iter_type put(iter_type s, bool intl, ios_base& f,
                char_type fill, const string_type& digits) const;
```
static locale::id id;

protected:
  ~money_put();
  virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                         long double units) const;
  virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                         const string_type& digits) const;
};

25.4.6.2.1 money_put 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).

25.4.6.2.2 money_put 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 \texttt{moneypunct<}\texttt{charT, Intl}> facet reference \texttt{mp} and the character mapping specified by a \texttt{ctype<}\texttt{charT}> facet reference \texttt{ct} obtained from the locale returned by \texttt{str.getloc()}, and \texttt{str.flags()}. The argument \texttt{units} is transformed into a sequence of wide characters as if by

\texttt{ct.widen(buf1, buf1 + sprintf(buf1, ".0Lf", units), buf2)}

for character buffers \texttt{buf1} and \texttt{buf2}. If the first character in \texttt{digits} or \texttt{buf2} is equal to \texttt{ct.widen('-')}, then the pattern used for formatting is the result of \texttt{mp.neg_format()}; otherwise the pattern is the result of \texttt{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 \texttt{digits} or \texttt{buf2}. In \texttt{digits}, only the optional leading minus sign and the immediately subsequent digit characters (as classified according to \texttt{ct}) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls \texttt{str.width(0)}.

2 Remarks: The currency symbol is generated if and only if (\texttt{str.flags()} \& \texttt{str.showbase}) is nonzero. If the number of characters generated for the specified format is less than the value returned by \texttt{str.width()} on entry to the function, then copies of \texttt{fill} are inserted as necessary to pad to the specified width. For the value \texttt{af} equal to (\texttt{str.flags()} \& \texttt{str.adjustfield}), if (\texttt{af == str.internal}) is \texttt{true}, the fill characters are placed where \texttt{none} or \texttt{space} appears in the formatting pattern; otherwise if (\texttt{af == str.left}) is \texttt{true}, they are placed after the other characters; otherwise, they are placed before the other characters. [\textit{Note:} It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using \texttt{num_get<>::get}. — end note]

3 Returns: An iterator pointing immediately after the last character produced.

25.4.6.3 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);
          
        };
  
}
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\(^\text{253}\). 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:

\[
\text{value ::= units [ decimal-point [ digits ]] | decimal-point digits if frac_digits() returns a positive value, or value ::= units otherwise.}
\]

The symbol `decimal-point` indicates the character returned by `decimal_point()`. The other symbols are defined as follows:

\[
\text{units ::= digits [ thousands-sep units ]}
\]

\[
\text{digits ::= adigit [ digits ]}
\]

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()`.

---

\(^{253}\) An array of `char`, rather than an array of `part`, is specified for `pattern::field` purely for efficiency.
25.4.6.3.1 moneypunct members

charT decimal_point() const;
charT thousands_sep() const;
string grouping() const;
string_type curr_symbol() const;
string_type positive_sign() const;
string_type negative_sign() const;
int frac_digits() const;

pattern pos_format() const;
pattern neg_format() const;

1 Each of these functions \( F \) returns the result of calling the corresponding virtual member function \( \text{do}_F() \).

25.4.6.3.2 moneypunct virtual functions

charT do_decimal_point() const;
1

Returns: The radix separator to use in case \( \text{do}_\text{frac_digits}() \) is greater than zero.\(^{254}\)

charT do_thousands_sep() const;
2

Returns: The digit group separator to use in case \( \text{do}_\text{grouping}() \) specifies a digit grouping pattern.\(^{255}\)

string do_grouping() const;
3

Returns: A pattern defined identically as, but not necessarily equal to, the result of \( \text{numpunct<\text{charT}}>::\text{do}_\text{grouping}() \).\(^{256}\)

string_type do_curr_symbol() const;
4

Returns: A string to use as the currency identifier symbol. \[ Note: \text{For specializations where the second template parameter is true, this is typically four characters long: a three-letter code as specified by ISO 4217 followed by a space. — end note] \)

string_type do_positive_sign() const;

string_type do_negative_sign() const;
5

Returns: \( \text{do}_\text{positive_sign}() \) returns the string to use to indicate a positive monetary value;\(^{257}\)

\( \text{do}_\text{negative_sign}() \) returns the string to use to indicate a negative value.

int do_frac_digits() const;
6

Returns: The number of digits after the decimal radix separator, if any.\(^{258}\)

pattern do_pos_format() const;

pattern do_neg_format() const;
7

Returns: The specializations required in Table 70 (25.3.1.1.1), namely \( \text{moneypunct<\text{char}}, \text{moneypunct<\text{wchar_t}}, \text{moneypunct<\text{char, true}}, \text{and moneypunct<\text{wchar_t, true}} \), return an object of type \( \text{pattern} \) initialized to \{ \text{symbol, sign, none, value} \}.\(^{259}\)

25.4.6.4 Class template moneypunct_byname

namespace std {


template<class charT, bool Intl = false>

class moneypunct_byname : public moneypunct<charT, Intl> {

public:

using pattern = money_base::pattern;

using string_type = basic_string<charT>;

explicit moneypunct_byname(const char*, size_t refs = 0);

explicit moneypunct_byname(const string&, size_t refs = 0);

254) In common U.S. locales this is ".".
255) In common U.S. locales this is ",".
256) To specify grouping by 3s, the value is "\003" not "3".
257) This is usually the empty string.
258) In common U.S. locales, this is 2.
259) Note that the international symbol returned by \( \text{do}_\text{curr_symbol}() \) usually contains a space, itself; for example, "USD ".

§ 25.4.6.4 744
protected:
    ~moneypunct_byname();
};
}

25.4.7 The message retrieval category

Class messages<\charT> implements retrieval of strings from message catalogs.

25.4.7.1 Class template messages

namespace std {
    class messages_base {
        public:
            using catalog = unspecified signed integer type;
    };

template<class charT>
    class messages : public locale::facet, public messages_base {
        public:
            using char_type = charT;
            using string_type = basic_string<charT>;
            explicit messages(size_t refs = 0);
            catalog open(const basic_string<char>& fn, const locale&) const;
            string_type get(catalog c, int set, int msgid,
                            const string_type& dfault) const;
            void close(catalog c) const;

            static locale::id id;
        protected:
            ~messages();
            virtual catalog do_open(const basic_string<char>& name, const locale& loc) const;
            virtual string_type do_get(catalog, int set, int msgid,
                                        const string_type& dfault) const;
            virtual void do_close(catalog) const;
    };
}

Values of type messages_base::catalog usable as arguments to members get and close can be obtained only by calling member open.

25.4.7.1.1 messages 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).

25.4.7.1.2 messages 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().

2 Returns a value less than 0 if no such catalog can be opened.

3 Remarks: The locale argument loc is used for character set code conversion when retrieving messages, if needed.
string_type do_get(catalog cat, int set, int msgid, const string_type& dfault) const;

4 Requires: cat shall be a catalog obtained from open() and not yet closed.

5 Returns: A message identified by arguments set, msgid, and dfault, according to an implementation-defined mapping. If no such message can be found, returns dfault.

void do_close(catalog cat) const;

6 Requires: cat shall be a catalog obtained from open() and not yet closed.

7 Effects: Releases unspecified resources associated with cat.

8 Remarks: The limit on such resources, if any, is implementation-defined.

25.4.7.2 Class template messages_byname

namespace std {
  template<class charT>
  class messages_byname : public messages<charT> {
public:
    using catalog = messages_base::catalog;
    using string_type = basic_string<charT>;

    explicit messages_byname(const char*, size_t refs = 0);
    explicit messages_byname(const string&, size_t refs = 0);

    protected:
      -messages_byname();
  };
}

25.4.8 Program-defined facets

A C++ program may define facets to be added to a locale and used identically as the built-in facets. To create a new facet interface, C++ programs simply derive from locale::facet a class containing a static member: static locale::id id.

[ Note: The locale member function templates verify its type and storage class. — end note ]

[ Example: Traditional global localization is still easy:

```cpp
#include <iostream>
#include <locale>

int main(int argc, char** argv) {
  using namespace std;
  locale::global(locale("")); // set the global locale
  cin.imbue(locale()); // imbue it on all the std streams
  cout.imbue(locale());
  cerr.imbue(locale());
  wcin.imbue(locale());
  wcout.imbue(locale());
  wcerr.imbue(locale());

  return MyObject(argc, argv).doit();
}
```
— end example ]

[ Example: Greater flexibility is possible:

```cpp
#include <iostream>
#include <locale>

int main() {
  using namespace std;
  cin.imbue(locale("")); // the user's preferred locale
  cout.imbue(locale::classic());
  double f;
  while (cin >> f) cout << f << endl;
```
In a European locale, with input 3.456,78, output is 3456.78. —end example

This can be important even for simple programs, which may need to write a data file in a fixed format, regardless of a user’s preference.

Example: Here is an example of the use of locales in a library interface.

```cpp
// file: Date.h
#include <iosfwd>
#include <string>
#include <locale>
class Date {
public:
    Date(unsigned day, unsigned month, unsigned year);
    std::string asString(const std::locale& = std::locale());
};
std::istream& operator>>(std::istream& s, Date& d);
std::ostream& operator<<(std::ostream& s, Date d);
```

This example illustrates two architectural uses of class `locale`.

The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale.

The second is in the operators `<<` and `>>`, where a locale “hitchhikes” on another object, in this case a stream, to the point where it is needed.

```cpp
// file: Date.C
#include "Date" // includes <ctime>
#include <sstream>
std::string Date::asString(const std::locale& l) {
    using namespace std;
    ostringstream s; s.imbue(l);
    s << *this; return s.str();
}
std::istream& operator>>(std::istream& s, Date& d) {
    using namespace std;
    istream::sentry cerberos(s);
    if (cerberos) {
        ios_base::iostate err = goodbit;
        struct tm t;
        use_facet<time_get<char>>(s.getloc()).get_date(s, 0, s, err, &t);
        if (!err) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
        s.setstate(err);
    }
    return s;
}
```

—end example

A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from `locale::facet`. The only member a C++ program must define is the static member `id`, which identifies your class interface as a new facet.

Example: Classifying Japanese characters:

```cpp
// file: <jctype>
#include <locale>
namespace My {
    using namespace std;
    class JCtype : public locale::facet {
public:
    static locale::id id; // required for use as a new locale facet
    bool is_kanji (wchar_t c) const;
}
```

§ 25.4.8
protected:
  ~JCtype() { }
};

// file: filt.C
#include <iostream>
#include <locale>
#include "jctype"     // above
std::locale::id My::JCtype::id; // the static JCtype member declared above.

int main() {
  using namespace std;
  using wctype = ctype<wchar_t>;
  locale loc(locale(""),     // the user's preferred locale ...
              new My::JCtype);     // and a new feature ...
  wchar_t c = use_facet<wctype>(loc).widen('!');
  if (!use_facet<My::JCtype>(loc).is_kanji(c))
    cout << "no it isn't!" << endl;
}

12 The new facet is used exactly like the built-in facets. —end example[

13 [ Example: Replacing an existing facet is even easier. The code does not define a member id because it is reusing the numpunct<charT> facet interface:

// file: my_bool.C
#include <iostream>
#include <locale>
#include <string>
namespace My {
  using namespace std;
  using cnumpunct = numpunct_byname<char>;
  class BoolNames : public cnumpunct {
    protected:
      string do_truename() const { return "Oui Oui!"; }
      string do_falsename() const { return "Mais Non!"; }
    BoolNames() { }
  public:
    BoolNames(const char* name) : cnumpunct(name) { }
  };
}

int main(int argc, char** argv) {
  using namespace std;
  // make the user's preferred locale, except for...
  locale loc(locale(""), new My::BoolNames(""));
  cout.imbue(loc);
  cout << boolalpha << "Any arguments today? " << (argc > 1) << endl;
}

—end example]

25.5 C library locales

25.5.1 Header <locale> synopsis

namespace std {

  struct lconv;

  char* setlocale(int category, const char* locale);
  lconv* localeconv();
}
The contents and meaning of the header `<locale>` are the same as the C standard library header `<locale.h>`.

Calls to the function `setlocale` may introduce a data race (20.5.5.9) with other calls to `setlocale` or with calls to the functions listed in Table 81.

See also: ISO C 7.11

### Table 81 — Potential `setlocale` data races

<table>
<thead>
<tr>
<th>fprintf</th>
<th>isprint</th>
<th>iswdigit</th>
<th>localeconv</th>
<th>tolower</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf</td>
<td>ispunct</td>
<td>iswgraph</td>
<td>mblen</td>
<td>toupper</td>
</tr>
<tr>
<td>isalnum</td>
<td>isisspace</td>
<td>iswlower</td>
<td>mbstowcs</td>
<td>towlower</td>
</tr>
<tr>
<td>isalpha</td>
<td>isupper</td>
<td>iswprint</td>
<td>mbtowc</td>
<td>towupper</td>
</tr>
<tr>
<td>isblank</td>
<td>iswalnum</td>
<td>iswpunct</td>
<td>setlocale</td>
<td>wcscoll</td>
</tr>
<tr>
<td>iscntrl</td>
<td>iswalpha</td>
<td>iswspace</td>
<td>strcoll</td>
<td>wcstod</td>
</tr>
<tr>
<td>isdigit</td>
<td>iswblank</td>
<td>iswupper</td>
<td>strerror</td>
<td>wcstombs</td>
</tr>
<tr>
<td>isgraph</td>
<td>iswcntrl</td>
<td>iswxdigit</td>
<td>strtod</td>
<td>wcscxfrm</td>
</tr>
<tr>
<td>islower</td>
<td>iswctype</td>
<td>isxdigit</td>
<td>strxfrm</td>
<td>wctomb</td>
</tr>
</tbody>
</table>
26 Containers library

26.1 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 82.

Table 82 — Containers library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.2 Requirements</td>
<td></td>
</tr>
<tr>
<td>26.3 Sequence containers</td>
<td>&lt;array&gt;, &lt;deque&gt;, &lt;forward_list&gt;, &lt;list&gt;, &lt;vector&gt;</td>
</tr>
<tr>
<td>26.4 Associative containers</td>
<td>&lt;map&gt;, &lt;set&gt;</td>
</tr>
<tr>
<td>26.5 Unordered associative containers</td>
<td>&lt;unordered_map&gt;, &lt;unordered_set&gt;</td>
</tr>
<tr>
<td>26.6 Container adaptors</td>
<td>&lt;queue&gt;, &lt;stack&gt;</td>
</tr>
</tbody>
</table>

26.2 Container requirements

26.2.1 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 (23.10.9.2), 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 83, 84, and 85 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 83 — Container requirements

<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>X::value_type</td>
<td>T</td>
<td></td>
<td>Requires: T is Erasable from X (see 26.2.1, below)</td>
<td>compile time</td>
</tr>
<tr>
<td>X::reference</td>
<td>T&amp;</td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_reference</td>
<td>const T&amp;</td>
<td></td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>Expression</td>
<td>Return type</td>
<td>Operational semantics</td>
<td>Assertion/note</td>
<td>Complexity</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------</td>
<td>------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>X::iterator</td>
<td>iterator type whose value type is T</td>
<td>any iterator category that meets the forward iterator requirements, convertible to X::const_iterator.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::const_iterator</td>
<td>constant iterator type whose value type is T</td>
<td>any iterator category that meets the forward iterator requirements.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::difference_type</td>
<td>signed integer type</td>
<td>is identical to the difference type of X::iterator and X::const_iterator.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::size_type</td>
<td>unsigned integer type</td>
<td>size_type can represent any non-negative value of difference_type.</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X u;</td>
<td>Postconditions:</td>
<td>u.empty()</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>X()</td>
<td>Postconditions:</td>
<td>X().empty()</td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>X(a)</td>
<td>Requires: T is CopyInsertable into X (see below).</td>
<td></td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>X u(a); X u = a;</td>
<td>Postconditions: a == X(a).</td>
<td></td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>X u(rv); X u = rv;</td>
<td>Postconditions: u shall be equal to the value that rv had before this construction.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = rv</td>
<td>All existing elements of a are either move assigned to or destroyed</td>
<td>a shall be equal to the value that rv had before this assignment</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>(&amp;a)-&gt;X()</td>
<td>void</td>
<td>the destructor is applied to every element of a; any memory obtained is deallocated.</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td>a.begin()</td>
<td>iterator; const_iterator for constant a</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
<tr>
<td>a.end()</td>
<td>iterator; const_iterator for constant a</td>
<td></td>
<td>constant</td>
<td></td>
</tr>
</tbody>
</table>
Table 83 — Container requirements (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.cbegin()</td>
<td>iterator</td>
<td>const_cast&lt;X&lt;br&gt;const&amp;&gt;(a).begin();</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.cend()</td>
<td>iterator</td>
<td>const_cast&lt;X&lt;br&gt;const&amp;&gt;(a).end();</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a == b</td>
<td>convertible to bool</td>
<td>== is an equivalence relation.&lt;br&gt;equal(a.begin(),&lt;br&gt;a.end(),&lt;br&gt;b.begin(),&lt;br&gt;b.end());</td>
<td>Requires: T is EqualityComparable&lt;br&gt;Constant if a.size() !=&lt;br&gt;b.size(),&lt;br&gt;linear otherwise</td>
<td>constant</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>exchanges the contents of a and b</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td>swap(a, b)</td>
<td>void</td>
<td>a.swap(b)</td>
<td>(Note A)</td>
<td></td>
</tr>
<tr>
<td>r = a</td>
<td>X&amp;</td>
<td>distance(a.begin(),&lt;br&gt;a.end())</td>
<td>Postconditions: r == a</td>
<td>linear</td>
</tr>
<tr>
<td>a.size()</td>
<td>size_type</td>
<td>distance(a.begin(),&lt;br&gt;a.end())</td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>a.max_size()</td>
<td>size_type</td>
<td>distance(begin(),&lt;br&gt;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() ==&lt;br&gt;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 28. —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

```plaintext
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 20.5.3.5). [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. Lvalues of any `Compare`, `Pred`, or `Hash` types belonging to `a` and `b` shall be swappable and shall be exchanged by calling `swap` as described in 20.5.3.2. If `allocator_traits<allocator_type>::propagate_on_container_swap::value` is `true`, then lvalues of type `allocator_type` shall be swappable and the allocators of `a` and `b` shall also be exchanged by calling `swap` as described in 20.5.3.2. 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 (27.2), the container is called `reversible` and satisfies the additional requirements in Table 84.

Table 84 — 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><code>X::reverse_iterator</code></td>
<td>iterator type whose value type is <code>T</code></td>
<td><code>reverse_iterator&lt;iterator&gt;</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>X::const_reverse_iterator</code></td>
<td>constant iterator type whose value type is <code>T</code></td>
<td><code>reverse_iterator&lt;const_iterator&gt;</code></td>
<td>compile time</td>
</tr>
<tr>
<td><code>a.rbegin()</code></td>
<td><code>reverse_iterator; const_reverse_iterator</code> for constant <code>a</code></td>
<td><code>reverse_iterator(end())</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>a.rend()</code></td>
<td><code>reverse_iterator; const_reverse_iterator</code> for constant <code>a</code></td>
<td><code>reverse_iterator(begin())</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>a.crbegin()</code></td>
<td><code>const_reverse_iterator</code></td>
<td><code>const_cast&lt;X const&amp;&gt;(a).rbegin()</code></td>
<td>constant</td>
</tr>
<tr>
<td><code>a.crend()</code></td>
<td><code>const_reverse_iterator</code></td>
<td><code>const_cast&lt;X const&amp;&gt;(a).rend()</code></td>
<td>constant</td>
</tr>
</tbody>
</table>

11 Unless otherwise specified (see 26.2.6.1, 26.2.7.1, 26.3.8.4, and 26.3.11.5) all container types defined in this Clause meet the following additional requirements:

11.1 — if an exception is thrown by an `insert()` or `emplace()` function while inserting a single element, that function has no effects.

11.2 — if an exception is thrown by a `push_back()`, `push_front()`, `emplace_back()`, or `emplace_front()` function, that function has no effects.

11.3 — no `erase()`, `clear()`, `pop_back()` or `pop_front()` function throws an exception.

11.4 — no copy constructor or assignment operator of a returned iterator throws an exception.

11.5 — no `swap()` function throws an exception.
— no **swap()** function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped. [Note: The **end()** iterator does not refer to any element, so it may be invalidated. — end note]

12 Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

13 A contiguous container is a container that supports random access iterators (27.2.7) and whose member types **iterator** and **const_iterator** are contiguous iterators (27.2.1).

14 Table 85 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 85 unless otherwise stated.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &lt; b</code></td>
<td>convertible to bool</td>
<td>lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())</td>
<td>Requires: <code>&lt;</code> is defined for values of <code>T</code>. <code>&lt;</code> is a total ordering relationship.</td>
<td>linear</td>
</tr>
<tr>
<td><code>a &gt; b</code></td>
<td>convertible to bool</td>
<td><code>b &lt; a</code></td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td>convertible to bool</td>
<td>!(a &gt; b)</td>
<td>linear</td>
<td></td>
</tr>
<tr>
<td><code>a &gt;= b</code></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 28. — end note]

15 All of the containers defined in this Clause and in 24.3.2 except **array** meet the additional requirements of an allocator-aware container, as described in Table 86.

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 **DefaultInsertable** into `X` means that the following expression is well-formed:

```cpp
allocator_traits<A>::construct(m, p)
```

(15.2) — An element of `X` is **default-inserted** if it is initialized by evaluation of the expression

```cpp
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 **MoveInsertable** into `X` means that the following expression is well-formed:

```cpp
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 **CopyInsertable** into `X` means that, in addition to `T` being **MoveInsertable** into `X`, the following expression is well-formed:

```cpp
allocator_traits<A>::construct(m, p, v)
```

and its evaluation causes the following postcondition to hold: The value of `v` is unchanged and is equivalent to `*p`.

§ 26.2.1
— \( T \) is **EmplaceConstructible into** \( X \) from \( \text{args} \), for zero or more arguments \( \text{args} \), means that the following expression is well-formed:

\[
\text{allocator_traits<A>::construct(m, p, args)}
\]

— \( T \) is **Erasable from** \( X \) means that the following expression is well-formed:

\[
\text{allocator_traits<A>::destroy(m, p)}
\]

[Note: A container calls \text{allocator_traits<A>::construct(m, p, 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 86, \( X \) 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 86 — Allocator-aware 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>allocator_type A</td>
<td></td>
<td>Requires: allocator_type::value_type is the same as ( X::\text{value_type} ).</td>
<td>compile time</td>
</tr>
<tr>
<td>get_allocator() A</td>
<td></td>
<td></td>
<td>constant</td>
</tr>
<tr>
<td>( X() ) ( X u; )</td>
<td></td>
<td>Requires: ( A ) is DefaultConstructible. Postconditions: ( u.\text{empty}() ) returns true, ( u.\text{get_allocator}() == \text{A()} )</td>
<td>constant</td>
</tr>
<tr>
<td>( X(m) ) ( X u(m); )</td>
<td></td>
<td>Postconditions: ( 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>Requires: ( T ) is CopyInsertable into ( X ). Postconditions: ( u == t, u.\text{get_allocator}() == m )</td>
<td>linear</td>
</tr>
<tr>
<td>( X(rv) ) ( X u(rv); )</td>
<td></td>
<td>Postconditions: ( u ) shall have the same elements as ( rv ) had before this construction; the value of ( u.\text{get_allocator}() ) shall be the same as the value of ( rv.\text{get_allocator}() ) before this construction.</td>
<td>constant if ( m == rv.\text{get_allocator}() ), otherwise linear</td>
</tr>
<tr>
<td>( X(rv, m) ) ( X u(rv, m); )</td>
<td></td>
<td>Requires: ( T ) is MoveInsertable into ( X ). Postconditions: ( u ) shall have 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 ) ( X&amp; )</td>
<td></td>
<td>Requires: ( T ) is CopyInsertable into ( X ) and CopyAssignable. Postconditions: ( a == t )</td>
<td>linear</td>
</tr>
</tbody>
</table>
Table 86 — Allocator-aware container 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>a = rv X&amp;</code></td>
<td><code>X&amp;</code></td>
<td>Requires: If <code>allocator_traits&lt;allocator_type&gt;::propagate_on_container_move_assignment::value</code> is false, <code>T</code> is <code>MoveInsertable</code> into <code>X</code> and <code>MoveAssignable</code>. All existing elements of <code>a</code> are either move assigned to or destroyed. Postconditions: <code>a</code> shall be equal to the value that <code>rv</code> had before this assignment.</td>
</tr>
<tr>
<td><code>a.swap(b)</code></td>
<td><code>void</code></td>
<td>exchanges the contents of <code>a</code> and <code>b</code></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:

(17.1) — The qualified-id `A::value_type` is valid and denotes a type (17.9.2).

(17.2) — The expression `declval<A&>().allocate(size_t{})` is well-formed when treated as an unevaluated operand.

26.2.2 Container data races [container.requirements.dataraces]

1 For purposes of avoiding data races (20.5.5.9), 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 20.5.5.9, 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]

26.2.3 Sequence containers [sequence.reqmts]

1 A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides four basic kinds of sequence containers: `vector`, `forward_list`, `list`, and `deque`. In addition, `array` is provided as a sequence container which provides limited sequence operations because it has a fixed number of elements. The library also provides container adaptors that make it easy to construct abstract data types, such as stacks or queues, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

2 The sequence containers offer the programmer different complexity trade-offs and should be used accordingly. `vector` or `array` is the type of sequence container that should be used by default. `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.

3 In Tables 87 and 88, `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 (17.9.2) 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`, `§ 26.2.3`
[i, j) denotes a valid range, il designates an object of type `initializer_list<value_type>`, n denotes a value of type `X::size_type`, p denotes a valid constant iterator to a, q denotes a valid dereferenceable constant iterator to a, [q1, q2) denotes a valid range of constant iterators in a, t denotes an lvalue or a const rvalue of `X::value_type`, and rv denotes a non-const rvalue of `X::value_type`. Args denotes a template parameter pack; args denotes a function parameter pack with the pattern `Args&&`.

The complexities of the expressions are sequence dependent.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X(n, t)</code></td>
<td></td>
<td>Requires: T shall be CopyInsertable into X. Postconditions: <code>distance(begin(), end())</code> == n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X(i, j)</code></td>
<td></td>
<td>Requires: T shall be EmplaceConstructible into X from *i. For vector, if the iterator does not meet the forward iterator requirements (27.2.5), T shall also be MoveInsertable into X. Each iterator in the range [i, j) shall be dereferenced exactly once. Postconditions: <code>distance(begin(), end())</code> == <code>distance(i, j)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>X(il)</code></td>
<td></td>
<td>Equivalent to <code>X(il.begin(), il.end())</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = il</code></td>
<td><code>X&amp;</code></td>
<td>Requires: T is CopyInsertable into X and CopyAssignable. Assigns the range [il.begin(), il.end()) into a. All existing elements of a are either assigned to or destroyed. Returns: *this.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.emplace(p, args)</code></td>
<td>iterator</td>
<td>Requires: T is EmplaceConstructible into X from args. For vector and deque, T is also MoveInsertable into X and MoveAssignable. Effects: Inserts an object of type T constructed with <code>std::forward&lt;Args&gt;(args)</code>... before p.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, t)</code></td>
<td>iterator</td>
<td>Requires: T shall be CopyInsertable into X. For vector and deque, T shall also be CopyAssignable. Effects: Inserts a copy of t before p.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, rv)</code></td>
<td>iterator</td>
<td>Requires: T shall be MoveInsertable into X. For vector and deque, T shall also be MoveAssignable. Effects: Inserts a copy of rv before p.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.insert(p, n, t)</code></td>
<td>iterator</td>
<td>Requires: T shall be CopyInsertable into X and CopyAssignable. Inserts n copies of t before p.</td>
</tr>
</tbody>
</table>
Table 87 — 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>a.insert(p,i,j)</td>
<td>iterator</td>
<td>Requires: T shall be EmplaceConstructible into X from *i. For vector and deque, T shall also be MoveInsertable into X, MoveConstructible, MoveAssignable, and swappable (20.5.3.2). Each iterator in the range [i, j) shall be dereferenced exactly once. Requires: i and j are not iterators into a. Inserts copies of elements in [i, j) before p.</td>
</tr>
<tr>
<td>a.insert(p, il)</td>
<td>iterator</td>
<td>a.insert(p, il.begin(), il.end()).</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the element pointed to by q.</td>
</tr>
<tr>
<td>a.erase(q1,q2)</td>
<td>iterator</td>
<td>Requires: For vector and deque, T shall be MoveAssignable. Effects: Erases the elements in the range [q1, q2).</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Destroys all elements in a. Invalidates all references, pointers, and iterators referring to the elements of a and may invalidate the past-the-end iterator. Postconditions: a.empty() returns true. Complexity: Linear.</td>
</tr>
<tr>
<td>a.assign(i,j)</td>
<td>void</td>
<td>Requires: T shall be EmplaceConstructible into X from *i and assignable from *i. For vector, if the iterator does not meet the forward iterator requirements (27.2.5), T shall also be MoveInsertable into X. Each iterator in the range [i, j) shall be dereferenced exactly once. Requires: i, j are not iterators into a. 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.</td>
</tr>
<tr>
<td>a.assign(il)</td>
<td>void</td>
<td>a.assign(il.begin(), il.end()).</td>
</tr>
<tr>
<td>a.assign(n,t)</td>
<td>void</td>
<td>Requires: T is not a reference into a. Replaces elements in a with n copies of t. Invalidates all references, pointers and iterators referring to the elements of a. For vector and deque, also invalidates the past-the-end iterator.</td>
</tr>
</tbody>
</table>

5 The iterator returned from a.insert(p, t) points to the copy of t inserted into a.

6 The iterator returned from a.insert(p, rv) points to the copy of rv inserted into a.

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.

8 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 24:

(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 88 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,</td>
</tr>
<tr>
<td></td>
<td>for constant a</td>
<td></td>
<td>array, deque,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>forward_list,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>list, vector</td>
</tr>
</tbody>
</table>
| `a.back()`     | reference; const_reference   | { auto tmp = a.end();
|                | for constant a               | --tmp;
|                |                              | return *tmp; }                                             | basic_string,   |
|                |                              |                                                           | array, deque,   |
|                |                              |                                                           | forward_list,   |
|                |                              |                                                           | list, vector    |
| `a.emplace_-  |
| front(args)`   | reference                    | Prepends an object of type T constructed with std::forward<Args>(
|                |                              | args)... Requires: T shall be EmplaceConstructible into X from args. |
|                |                              | Returns: a.front().                                       | deque,          |
|                |                              |                                                           | forward_list,   |
|                |                              |                                                           | list           |
Table 88 — Optional sequence container operations (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.emplace_back(args)</td>
<td>reference</td>
<td>Appends an object of type ( T ) constructed with ( \text{std}::\text{forward&lt;Args&gt;() args} ).... Requires: ( T ) shall be ( \text{EmplaceConstructible} ) into ( X ) from ( \text{args} ). For ( \text{vector} ), ( T ) shall also be ( \text{MoveInsertable} ) into ( X ). Returns: ( \text{a.back()} ).</td>
<td>deque, list, vector</td>
</tr>
<tr>
<td>a.push_front(t)</td>
<td>void</td>
<td>Prepends a copy of ( t ). Requires: ( T ) shall be ( \text{CopyInsertable} ) into ( X ).</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td>a.push_front(rv)</td>
<td>void</td>
<td>Prepends a copy of ( rv ). Requires: ( T ) shall be ( \text{MoveInsertable} ) into ( X ).</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td>a.push_back(t)</td>
<td>void</td>
<td>Appends a copy of ( t ). Requires: ( T ) shall be ( \text{CopyInsertable} ) into ( X ).</td>
<td>basic_string, deque, list, vector</td>
</tr>
<tr>
<td>a.push_back(rv)</td>
<td>void</td>
<td>Appends a copy of ( rv ). Requires: ( T ) shall be ( \text{MoveInsertable} ) into ( X ).</td>
<td>basic_string, deque, list, vector</td>
</tr>
<tr>
<td>a.pop_front()</td>
<td>void</td>
<td>Destroys the first element. Requires: ( \text{a.empty()} ) shall be false.</td>
<td>deque, forward_list, list</td>
</tr>
<tr>
<td>a.pop_back()</td>
<td>void</td>
<td>Destroys the last element. Requires: ( \text{a.empty()} ) shall be false.</td>
<td>basic_string, deque, list, vector</td>
</tr>
<tr>
<td>a[n]</td>
<td>reference; const_reference</td>
<td>*(a.begin() + n)</td>
<td>basic_string, array, deque, vector</td>
</tr>
<tr>
<td>a.at(n)</td>
<td>reference; const_reference</td>
<td>*(a.begin() + n)</td>
<td>basic_string, array, deque, vector</td>
</tr>
</tbody>
</table>

15 The member function \( \text{at()} \) provides bounds-checked access to container elements. \( \text{at()} \) throws \text{out_of_range} \text{if} \( n >= \text{a.size()} \).

26.2.4 Node handles

26.2.4.1 node_handle overview

1 A \textit{node handle} is an object that accepts ownership of a single element from an associative container (26.2.6) or an unordered associative container (26.2.7). 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 89.

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 \textit{node_handle} is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name.

4 If a user-defined specialization of \textit{pair} exists for \textit{pair<const Key, T>} or \textit{pair<Key, T>}, where \textit{Key} is the container’s \textit{key_type} and \textit{T} is the container’s \textit{mapped_type}, the behavior of operations involving node handles is undefined.
Table 89 — Container types with compatible nodes

<table>
<thead>
<tr>
<th>Container Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>map&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>map&lt;K, T, C1, A&gt;</td>
<td>multimap&lt;K, T, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>set&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>set&lt;K, C1, A&gt;</td>
<td>multiset&lt;K, C2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, T, H1, E1, A&gt;</td>
<td>unordered_map&lt;K, T, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_map&lt;K, T, H1, E1, A&gt;</td>
<td>unordered_multimap&lt;K, T, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_set&lt;K, H1, E1, A&gt;</td>
<td>unordered_set&lt;K, H2, E2, A&gt;</td>
</tr>
<tr>
<td>unordered_set&lt;K, H1, E1, A&gt;</td>
<td>unordered_multiset&lt;K, H2, E2, A&gt;</td>
</tr>
</tbody>
</table>

template<unspeficied>
class node_handle {
    public:
        // These type declarations are described in Tables 90 and 91.
        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::rebind_traits<container_node_type>::pointer ptr_;
        optional<allocator_type> alloc_;

    public:
        constexpr node_handle() noexcept : ptr_(), alloc_() {}
        ~node_handle();
        node_handle(node_handle&&) noexcept;
        node_handle& operator=(node_handle&&);
        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;

        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);
        }
    }

26.2.4.2 node_handle constructors, copy, and assignment [container.node.cons]

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 Requires: Either !alloc_ or ator_traits::propagate_on_container_move_assignment is true, or alloc_ == nh.alloc_.

Effects:
— If \( \text{ptr}_\_ \neq \text{nullptr} \), destroys the \text{value_type} subobject in the container_node_type object pointed to by \( \text{ptr}_\_ \) by calling \text{ator_traits}::\text{destroy}, then deallocates \( \text{ptr}_\_ \) by calling \text{ator_traits}::\text{rebind_traits<container_node_type>::deallocate}.

(3.2) Assigns \text{nh}.\text{ptr}_\_ to \text{ptr}_\_.

(3.3) If \( \text{alloc}_\_ \) or \text{ator_traits}::\text{propagate_on_container_move_assignment} is true, move assigns \text{nh}.\text{alloc}_\_ to \text{alloc}_\_.

(3.4) Assigns \text{nullptr} to \text{nh}.\text{ptr}_\_ and assigns \text{nullopt} to \text{nh}.\text{alloc}_\_.

Returns: \*\text{this}.

Throws: Nothing.

### 26.2.4.3 node_handle destructor

\~\text{node\_handle}();

Effects: If \( \text{ptr}_\_ \neq \text{nullptr} \), destroys the \text{value_type} subobject in the container_node_type object pointed to by \( \text{ptr}_\_ \) by calling \text{ator_traits}::\text{destroy}, then deallocates \( \text{ptr}_\_ \) by calling \text{ator_traits}::\text{rebind_traits<container_node_type>::deallocate}.

### 26.2.4.4 node_handle observers

\text{value_type}\& value() const;

Requires: \text{empty()} == false.

Returns: A reference to the \text{value_type} subobject in the container_node_type object pointed to by \text{ptr}_\_.

Throws: Nothing.

\text{key_type}\& key() const;

Requires: \text{empty()} == false.

Returns: A non-const reference to the \text{key_type} member of the \text{value_type} subobject in the container_node_type object pointed to by \text{ptr}_\_.

Throws: Nothing.

Remarks: Modifying the key through the returned reference is permitted.

\text{mapped_type}\& mapped() const;

Requires: \text{empty()} == false.

Returns: A reference to the \text{mapped_type} member of the \text{value_type} subobject in the container_node_type object pointed to by \text{ptr}_\_.

Throws: Nothing.

\text{allocator_type} get_allocator() const;

Requires: \text{empty()} == false.

Returns: \*\text{alloc}_\_.

Throws: Nothing.

explicit operator bool() const noexcept;

Returns: \text{ptr}_\_ \neq \text{nullptr}.

[[\text{nodiscard}]] bool empty() const noexcept;

Returns: \text{ptr}_\_ == \text{nullptr}.

### 26.2.4.5 node_handle modifiers

\text{void}\ swap(\text{node\_handle}\& \text{nh})

\text{nothrow}(@text{ator\_traits}::\text{propagate\_on\_container\_swap::value} ||
The associative containers with unique keys and the unordered containers with unique keys have a member function \texttt{insert} that returns a nested type \texttt{insert_return_type}. That return type is a specialization of the type specified in this subclause.

\begin{verbatim}
    template<class Iterator, class NodeType>
    struct INSERT_RETURN_TYPE
    {
        Iterator position;
        bool inserted;
        NodeType node;
    };
\end{verbatim}

The name \texttt{INSERT_RETURN_TYPE} is exposition only. \texttt{INSERT_RETURN_TYPE} has the template parameters, data members, and special members specified above. It has no base classes or members other than those specified.

The associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: \texttt{set}, \texttt{multiset}, \texttt{map} and \texttt{multimap}.

Each associative container is parameterized on \texttt{Key} and an ordering relation \texttt{Compare} that induces a strict weak ordering \((28.7)\) on elements of \texttt{Key}. In addition, \texttt{map} and \texttt{multimap} associate an arbitrary \texttt{mapped type} \texttt{T} with the \texttt{Key}. The object of type \texttt{Compare} is called the \texttt{comparison object} of a container.

The phrase “equivalence of keys” means the equivalence relation imposed by the comparison and \texttt{not} the \texttt{operator==} on keys. That is, two keys \texttt{k1} and \texttt{k2} are considered to be equivalent if for the comparison object \texttt{comp, comp(k1, k2) == false \&\& comp(k2, k1) == false}. For any two keys \texttt{k1} and \texttt{k2} in the same container, calling \texttt{comp(k1, k2)} shall always return the same value.

An associative container supports \texttt{unique keys} if it may contain at most one element for each key. Otherwise, it supports \texttt{equivalent keys}. The \texttt{set} and \texttt{map} classes support unique keys; the \texttt{multiset} and \texttt{multimap} classes support equivalent keys. For \texttt{multiset} and \texttt{multimap}, \texttt{insert, emplace, and erase} preserve the relative ordering of equivalent elements.

For \texttt{set} and \texttt{multiset} the value type is the same as the key type. For \texttt{map} and \texttt{multimap} it is equal to \texttt{pair<const Key, T>}.

\texttt{iterator} of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both \texttt{iterator} and \texttt{const_iterator} are constant iterators. It is unspecified whether or not \texttt{iterator} and \texttt{const_iterator} are the same type. \[\text{Note: \texttt{iterator} and \texttt{const_iterator} have identical semantics in this case, and \texttt{iterator} is convertible to \texttt{const_iterator}. Users can avoid violating the one-definition rule by always using \texttt{const_iterator} in their function parameter lists.} \end{note}

The associative containers meet all the requirements of Allocator-aware containers \((26.2.1)\), except that for \texttt{map} and \texttt{multimap}, the requirements placed on \texttt{value_type} in Table \ref{input_iterator_requirements} apply instead to \texttt{key_type} and \texttt{mapped_type}. \[\text{Note: For example, in some cases \texttt{key_type} and \texttt{mapped_type} are required to be \texttt{CopyAndAssignable} even though the associated \texttt{value_type, pair<const key_type, mapped_type>}, is not \texttt{CopyAndAssignable.} \end{note}

In Table \ref{associative_requirements}, \texttt{X} denotes an associative container class, \texttt{a} denotes a value of type \texttt{X}, \texttt{a2} denotes a value of a type with nodes compatible with type \texttt{X} \texttt{(Table 89)}, \texttt{b} denotes a possibly \texttt{const} value of type \texttt{X}, \texttt{u} denotes the name of a variable being declared, \texttt{a\_uniq} denotes a value of type \texttt{X} when \texttt{X} supports unique keys, \texttt{a\_eq} denotes a value of type \texttt{X} when \texttt{X} supports multiple keys, \texttt{a\_tran} denotes a possibly \texttt{const} value of type \texttt{X} when the \texttt{qualified-id} \texttt{X::key\_compare::is\_transparent} is valid and denotes a type \texttt{(17.9.2)}, \texttt{I} and \texttt{J} satisfy input iterator requirements and refer to elements implicitly convertible to \texttt{value_type}, \texttt{[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}, etc.

\begin{note}
\texttt{iterator_traits::is\_always\_equal::value);}

\begin{verbatim}
Requires: !alloc_ || !nh.alloc_ || \texttt{ator_traits::propagate\_on\_container\_swap} is true, or alloc_ == nh.alloc_.
\end{verbatim}

\begin{verbatim}
Effects: Calls \texttt{swap(ptr_, nh.ptr_)}. If !alloc_ || !nh.alloc_ || \texttt{ator_traits::propagate\_on\_container\_swap} is true calls \texttt{swap(alloc_, nh.alloc_)}.
\end{verbatim}

\begin{verbatim}
26.2.5 Insert return type \hspace{1cm} [container.insert.return]
\end{verbatim}

\begin{verbatim}
The associative containers with unique keys and the unordered containers with unique keys have a member function \texttt{insert} that returns a nested type \texttt{insert\_return\_type}. That return type is a specialization of the type specified in this subclause.
\end{verbatim}

\begin{verbatim}
Insertion of a value uses \texttt{insert\_return\_type}.
\end{verbatim}

\begin{verbatim}
26.2.6 Associative containers \hspace{1cm} [associative.reqmts]
\end{verbatim}

\begin{verbatim}
Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: \texttt{set}, \texttt{multiset}, \texttt{map} and \texttt{multimap}.
\end{verbatim}

\begin{verbatim}
Each associative container is parameterized on \texttt{Key} and an ordering relation \texttt{Compare} that induces a strict weak ordering \((28.7)\) on elements of \texttt{Key}. In addition, \texttt{map} and \texttt{multimap} associate an arbitrary \texttt{mapped type} \texttt{T} with the \texttt{Key}. The object of type \texttt{Compare} is called the \texttt{comparison object} of a container.
\end{verbatim}

\begin{verbatim}
The phrase “equivalence of keys” means the equivalence relation imposed by the comparison and \texttt{not} the \texttt{operator==} on keys. That is, two keys \texttt{k1} and \texttt{k2} are considered to be equivalent if for the comparison object \texttt{comp, comp(k1, k2) == false \&\& comp(k2, k1) == false}. For any two keys \texttt{k1} and \texttt{k2} in the same container, calling \texttt{comp(k1, k2)} shall always return the same value.
\end{verbatim}

\begin{verbatim}
An associative container supports \texttt{unique keys} if it may contain at most one element for each key. Otherwise, it supports \texttt{equivalent keys}. The \texttt{set} and \texttt{map} classes support unique keys; the \texttt{multiset} and \texttt{multimap} classes support equivalent keys. For \texttt{multiset} and \texttt{multimap}, \texttt{insert, emplace, and erase} preserve the relative ordering of equivalent elements.
\end{verbatim}

\begin{verbatim}
For \texttt{set} and \texttt{multiset} the value type is the same as the key type. For \texttt{map} and \texttt{multimap} it is equal to \texttt{pair<const Key, T>}.
\end{verbatim}

\begin{verbatim}
\texttt{iterator} of an associative container is of the bidirectional iterator category. For associative containers where the value type is the same as the key type, both \texttt{iterator} and \texttt{const\_iterator} are constant iterators. It is unspecified whether or not \texttt{iterator} and \texttt{const\_iterator} are the same type. \[\text{Note: \texttt{iterator} and \texttt{const\_iterator} have identical semantics in this case, and \texttt{iterator} is convertible to \texttt{const\_iterator}. Users can avoid violating the one-definition rule by always using \texttt{const\_iterator} in their function parameter lists.} \end{note}

\begin{verbatim}
The associative containers meet all the requirements of Allocator-aware containers \((26.2.1)\), except that for \texttt{map} and \texttt{multimap}, the requirements placed on \texttt{value\_type} in Table \ref{input_iterator_requirements} apply instead to \texttt{key\_type} and \texttt{mapped\_type}. \[\text{Note: For example, in some cases \texttt{key\_type} and \texttt{mapped\_type} are required to be \texttt{Copy\&Assign\_able} even though the associated \texttt{value\_type, pair<const key\_type, mapped\_type>}, is not \texttt{Copy\&Assign\_able.} \end{note}

\begin{verbatim}
In Table \ref{associative_requirements}, \texttt{X} denotes an associative container class, \texttt{a} denotes a value of type \texttt{X}, \texttt{a2} denotes a value of a type with nodes compatible with type \texttt{X} \texttt{(Table 89)}, \texttt{b} denotes a possibly \texttt{const} value of type \texttt{X}, \texttt{u} denotes the name of a variable being declared, \texttt{a\_uniq} denotes a value of type \texttt{X} when \texttt{X} supports unique keys, \texttt{a\_eq} denotes a value of type \texttt{X} when \texttt{X} supports multiple keys, \texttt{a\_tran} denotes a possibly \texttt{const} value of type \texttt{X} when the \texttt{qualified-id} \texttt{X::key\_compare::is\_transparent} is valid and denotes a type \texttt{(17.9.2)}, \texttt{I} and \texttt{J} satisfy input iterator requirements and refer to elements implicitly convertible to \texttt{value\_type}, \texttt{[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}, etc.
\end{verbatim}

\section*{26.2.6}

\[763\]
r denotes a valid dereferenceable iterator to a, \([q_1, q_2)\) denotes a valid range of constant iterators in a, il designates an object of type `initializer_list<value_type>`, t denotes a value of type X::value_type, k denotes a value of type X::key_type and c denotes a possibly const value of type X::key_compare; kl is a value such that a is partitioned (28.7) with respect to c(r, kl), with r the key value of e and e in a; ku is a value such that a is partitioned with respect to !c(ku, r); ke is a value such that a is partitioned with respect to c(r, ke) and !c(ke, r), with c(r, ke) implying !c(ke, r). A denotes the storage allocator used by X, if any, or `allocator<X::value_type>` otherwise, m denotes an allocator of a type convertible to A, and nh denotes a non-const rvalue of type X::node_type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>X::key_type</td>
<td>Key</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::mapped_type (map and multimap only)</td>
<td>T</td>
<td>compile time</td>
<td></td>
</tr>
<tr>
<td>X::value_type (set and multiset only)</td>
<td>Key</td>
<td>Requires: value_type is Erasable from X</td>
<td></td>
</tr>
<tr>
<td>X::value_type (map and multimap only)</td>
<td>pair&lt;const Key, T&gt;</td>
<td>Requires: value_type is Erasable from X</td>
<td></td>
</tr>
<tr>
<td>X::key_compare</td>
<td>Compare</td>
<td>Requires: key_compare is CopyConstructible.</td>
<td></td>
</tr>
<tr>
<td>X::value_compare</td>
<td>a binary predicate type</td>
<td>is the same as key_compare for set and multiset; is an ordering relation on pairs induced by the first component (i.e., Key) for map and multimap.</td>
<td></td>
</tr>
<tr>
<td>X::node_type</td>
<td>a specialization of a node_handle class template, such that the public nested types are the same types as the corresponding types in X.</td>
<td>see 26.2.4</td>
<td></td>
</tr>
<tr>
<td>X(c)</td>
<td>Effects: Constructs an empty constant container. Uses a copy of c as a comparison object.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u(c);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X()</td>
<td>Requires: key_compare is DefaultConstructible. Effects: Constructs an empty container. Uses Compare() as a comparison object.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X u;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 90 — Associative container requirements (in addition to container)
Table 90 — 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(i,j,c)</td>
<td></td>
<td>Requires: value_type is EmplaceConstructible into X from i. Effects: Constructs an empty container and inserts elements from the range [i, j) into it; uses c as a comparison object.</td>
<td>N log N in general, where N has the value distance(i, j); linear if [i, j) is sorted with value_comp()</td>
</tr>
<tr>
<td>X u(i,j,c);</td>
<td></td>
<td></td>
<td>same as above</td>
</tr>
<tr>
<td>X(i,j)</td>
<td></td>
<td>Requires: key_compare is DefaultConstructible. value_type is EmplaceConstructible into X from i. Effects: Same as above, but uses Compare() as a comparison object.</td>
<td>same as above</td>
</tr>
<tr>
<td>X u(i,j);</td>
<td></td>
<td></td>
<td>same as above</td>
</tr>
<tr>
<td>X(il)</td>
<td></td>
<td>same as X(il.begin(), il.end())</td>
<td>same as X(il.begin(), il.end())</td>
</tr>
<tr>
<td>X(il,c)</td>
<td></td>
<td>same as X(il.begin(), il.end(), c)</td>
<td>same as X(il.begin(), il.end(), c)</td>
</tr>
<tr>
<td>a = il X&amp;</td>
<td>X::key_compare()</td>
<td>Requires: value_type is CopyInsertable into X and CopyAssignable. Effects: Assigns the range [il.begin(), il.end()) to a. All existing elements of a are either assigned to or destroyed.</td>
<td>N log N in general, where N has the value il.size() + a.size(); linear if [il.begin(), il.end()) is sorted with value_comp()</td>
</tr>
<tr>
<td>b.key_compare()</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_compare()</td>
<td>X::value_compare()</td>
<td>returns an object of value_compare constructed out of the comparison object</td>
<td>constant</td>
</tr>
<tr>
<td>a_uniq. emplace()</td>
<td>pair&lt; iterator, bool&gt;</td>
<td>Requires: value_type shall be EmplaceConstructible 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>
</tbody>
</table>
### Table 90 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_eq. emplace(args)</td>
<td>iterator</td>
<td>Requires: value_type shall be EmplaceConstructible into X from args.</td>
<td>logarithmic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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></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>Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable 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>
<tr>
<td>a_eq. insert(t)</td>
<td>iterator</td>
<td>Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable into X. Effects: Inserts t and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to t exists in a_eq, t is inserted at the end of that range.</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 90 — 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, iterator t)</td>
<td>iterator</td>
<td>Requires: If t is a non-const rvalue expression, value_type shall be MoveInsertable into X; otherwise, value_type shall be CopyInsertable 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, void j)</td>
<td>void</td>
<td>Requires: value_type shall be EmplaceConstructible into X from *i. Requires: i, j are not iterators into a. 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(\text{a.size()} + N), where N has the value distance(i, j)</td>
</tr>
<tr>
<td>a.insert( void il)</td>
<td>equivalent to a.insert(il.begin(), il.end())</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_uniq. insert_(return_type nh)</td>
<td>return_type</td>
<td>Requires: nh is empty or a_uniq.get_allocator() == nh.get_allocator(). Effects: If nh is empty, has no effect. Otherwise, inserts the element owned by nh if and only if there is no element in the container with a key equivalent to nh.key(). Postconditions: If nh is empty, inserted is false, position is end(), and node is empty. Otherwise if the insertion took place, inserted is true, position points to the inserted element, and node is empty; if the insertion failed, inserted is false, node has the previous value of nh, and position points to an element with a key equivalent to nh.key().</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>
Table 90 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_eq. insert(nh)</td>
<td>iterator</td>
<td>Requires: nh is empty or a_eq.get_allocator() == nh.get_allocator().</td>
<td>logarithmic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Effects: If nh is empty, has no effect and returns a_eq.end().</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise, inserts the element owned by nh and returns an iterator pointing to</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>the newly inserted element. If a range containing elements with keys</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>equivalent to nh.key() exists in a_eq, the element is inserted at the end of</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>that range.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Postconditions: nh is empty.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>a.insert(p, nh)                   iterator</td>
<td>logarithmic in general, but</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires: nh is empty or a.get_allocator() == nh.get Allocator().</td>
<td>amortized constant if the element is inserted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Effects: If nh is empty, has no effect and returns a.end().</td>
<td>right before p.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Otherwise, inserts the element owned by nh if and only if there is no element</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>containing elements with keys equivalent to nh.key() in containers with unique</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>keys; always inserts the element owned by nh in containers with equivalent</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>keys. Always returns the iterator pointing to the element with key equivalent</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>to nh.key(). The element is inserted as close as possible to the position</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>just prior to p.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Postconditions: nh is empty if insertion succeeds, unchanged if insertion</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fails.</td>
<td></td>
</tr>
<tr>
<td>a.extract(k)</td>
<td>node_type</td>
<td>removes the first element in the container with key equivalent to k. Returns</td>
<td>log(a.size())</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a node_type owning the element if found, otherwise an empty node_type.</td>
<td></td>
</tr>
<tr>
<td>a.extract(q)</td>
<td>node_type</td>
<td>removes the element pointed to by q. Returns a node_type owning that element.</td>
<td>amortized constant</td>
</tr>
</tbody>
</table>
### Table 90 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion / note</th>
<th>Complexity</th>
</tr>
</thead>
</table>
| `a.merge(a2)` | `void` | Requires: `a.get_allocator()` == `a2.get_allocator()`. Attempts to extract each element in `a2` and insert it into `a` using the comparison object of `a`. In containers with unique keys, if there is an element in `a` with key equivalent to the key of an element from `a2`, then that element is not extracted from `a2`. **Postconditions**: Pointers and references to the transferred elements of `a2` refer to those same elements but as members of `a`. Iterators referring to the transferred elements will continue to refer to their elements, but they now behave as iterators into `a`, not into `a2`. **Throws**: Nothing unless the comparison object throws. | $N \log(a.size() + N)$, where $N$ has the value `a2.size()`.

| `a.erase(k)` | `size_type` | erases all elements in the container with key equivalent to `k`. returns the number of erased elements. | $\log(a.size()) + a.count(k)$ |

| `a.erase(q)` | `iterator` | erases the element pointed to by `q`. Returns an iterator pointing to the element immediately following `q` prior to the element being erased. If no such element exists, returns `a.end()` | amortized constant |

| `a.erase(r)` | `iterator` | erases the element pointed to by `r`. Returns an iterator pointing to the element immediately following `r` prior to the element being erased. If no such element exists, returns `a.end()` | amortized constant |

| `a.erase(q1, q2)` | `iterator` | erases all the elements in the range `[q1, q2)`. Returns an iterator pointing to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned. | $\log(a.size()) + N$, where $N$ has the value `distance(q1, q2)` |

| `a.clear()` | `void` | `a.erase(a.begin(), a.end())` | linear in `a.size()` |

*Postconditions: `a.empty()` returns `true`.**
Table 90 — Associative container requirements (in addition to container) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b.find(k)</code></td>
<td>iterator; const_iterator for constant b.</td>
<td>returns an iterator pointing to an element with the key equivalent to k, or <code>b.end()</code> if such an element is not found</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a_tran.find(ke)</code></td>
<td>iterator; const_iterator for constant a_tran</td>
<td>returns an iterator pointing to an element with key <code>r</code> such that <code>!c(r, ke) &amp;&amp; !c(ke, r)</code>, or <code>a_tran.end()</code> if such an element is not found</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>b.count(k)</code></td>
<td>size_type</td>
<td>returns the number of elements with key equivalent to k</td>
<td>log(<code>b.size()</code>) + b.count(k)</td>
</tr>
<tr>
<td><code>a_tran.count(ke)</code></td>
<td>size_type</td>
<td>returns the number of elements with key <code>r</code> such that <code>!c(r, ke) &amp;&amp; !c(ke, r)</code></td>
<td>log(<code>a_tran.size()</code>) + a_tran.count(ke)</td>
</tr>
<tr>
<td><code>b.lower_bound(k)</code></td>
<td>iterator; const_iterator for constant b.</td>
<td>returns an iterator pointing to the first element with key not less than k, or <code>b.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a_tran.lower_bound(kl)</code></td>
<td>iterator; const_iterator for constant a_tran</td>
<td>returns an iterator pointing to the first element with key <code>r</code> such that <code>!c(r, kl)</code>, or <code>a_tran.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>b.upper_bound(k)</code></td>
<td>iterator; const_iterator for constant b.</td>
<td>returns an iterator pointing to the first element with key greater than k, or <code>b.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a_tran.upper_bound(ku)</code></td>
<td>iterator; const_iterator for constant a_tran</td>
<td>returns an iterator pointing to the first element with key <code>r</code> such that <code>c(ku, r)</code>, or <code>a_tran.end()</code> if such an element is not found.</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>b.equal_range(k)</code></td>
<td><code>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for constant b.</code></td>
<td>equivalent to make_pair(b.lower_bound(k), b.upper_bound(k)).</td>
<td>logarithmic</td>
</tr>
<tr>
<td><code>a_tran.equal_range(ke)</code></td>
<td><code>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for constant a_tran.</code></td>
<td>equivalent to make_pair(a_tran.lower_bound(ke), a_tran.upper_bound(ke)).</td>
<td>logarithmic</td>
</tr>
</tbody>
</table>

9 The `insert` and `emplace` members shall not affect the validity of iterators and references to the container, and the `erase` members shall invalidate only iterators and references to the erased elements.
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.

The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive, the following condition holds:

\[ \text{value_comp}(\ast j, \ast i) = \text{false} \]

For associative containers with unique keys the stronger condition holds:

\[ \text{value_comp}(\ast i, \ast j) \neq \text{false} \]

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, either through 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.

The member function templates `find`, `count`, `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 (17.9.2).

A deduction guide for an associative container shall not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Compare` template parameter and a type that qualifies as an allocator is deduced for that parameter.

### 26.2.6.1 Exception safety guarantees

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).

### 26.2.7 Unordered associative containers

1. Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.
2. Unordered associative containers conform to the requirements for Containers (26.2), except that the expressions `a == b` and `a != b` have different semantics than for the other container types.
3. Each unordered associative container is parameterized by `Key`, by a function object type `Hash` that meets the `Hash` requirements (20.5.3.4) 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`.
4. The container’s object of type `Hash` — denoted by `hash` — is called the `hash function` of the container. The container’s object of type `Pred` — denoted by `pred` — is called the `key equality predicate` of the container.
5. Two values `k1` and `k2` of type `Key` are considered equivalent if the container’s key equality predicate 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 \(k_2\) in the same container, calling \(\text{pred}(k_1, k_2)\) shall always return the same value. For any key \(k\) in a container, calling \(\text{hash}(k)\) 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 (26.2.1), except that for unordered_map and unordered_multimap, the requirements placed on value_type in Table 83 apply instead to key_type and mapped_type. [Note: For example, key_type and mapped_type are sometimes required to be CopyAssignable even though the associated value_type, pair<const key_type, mapped_type>, is not CopyAssignable. —end note]

In Table 91: \(X\) denotes an unordered associative container class, \(a\) denotes a value of type \(X\), \(a_2\) denotes a value of a type with nodes compatible with type \(X\) (Table 89), \(b\) 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, \(i\) and \(j\) denote input iterators that refer to value_type, \([i, j]\) denotes a valid range, \(p\) and \(q_2\) denote valid constant iterators to \(a\), \(q\) and \(q_1\) denote valid dereferenceable constant iterators to \(a\), \(r\) denotes a valid dereferenceable iterator to \(a\), \([q_1, q_2]\) denotes a valid range in \(a\), \(a_i\) denotes a value of type initializer_list<value_type>, \(t\) denotes a value of type \(X::value_type\), \(h\) denotes a value of type key_equal, \(n\) denotes a value of type size_type, \(z\) denotes a value of type float, and \(nh\) denotes a non-const value of type \(X::node_type\).

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X::key_type)</td>
<td>Key</td>
<td></td>
<td>compile time</td>
</tr>
<tr>
<td>(X::mapped_type) (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>Requires: value_type is Erasable from X</td>
<td>compile time</td>
</tr>
</tbody>
</table>
Table 91 — 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::value_type</td>
<td>pair&lt;const Key, T&gt;</td>
<td>Requires: value_type is Erasable from X</td>
<td>compile time</td>
</tr>
<tr>
<td>X::hasher</td>
<td>Hash</td>
<td>Hash shall be a unary function object type such that the expression hf(k) has type size_t.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::key_equal</td>
<td>Pred</td>
<td>Requires: Pred is CopyConstructible. Pred shall be a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::local_iterator</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::iterator’s.</td>
<td>A local_iterator object may be used to iterate through a single bucket, but may not be used to iterate across buckets.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::const_local_iterator</td>
<td>An iterator type whose category, value type, difference type, and pointer and reference types are the same as X::const_iterator’s.</td>
<td>A const_local_iterator object may be used to iterate through a single bucket, but may not be used to iterate across buckets.</td>
<td>compile time</td>
</tr>
<tr>
<td>X::node_type</td>
<td>a specialization of a node_handle class template, such that the public nested types are the same types as the corresponding types in X.</td>
<td>see 26.2.4</td>
<td>compile time</td>
</tr>
</tbody>
</table>

X(n, hf, eq)  
X a(n, hf, eq);  
X(n, hf)  
X a(n, hf);  
X(n)  
X a(n);  

Effects: Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate.  
\( O(n) \)

\$ 26.2.7 \$
<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</td>
<td>Requires: hasher and key_equal are DefaultConstructible. 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.</td>
<td>constant</td>
</tr>
<tr>
<td>X a;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n, hf, eq)</td>
<td>X</td>
<td>Requires: value_type is EmplaceConstructible 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 ( O(N) ) (( N ) is distance(i, j)), worst case ( O(N^2) )</td>
</tr>
<tr>
<td>X a(i, j, n, hf, eq);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n, hf)</td>
<td>X</td>
<td>Requires: key_equal is DefaultConstructible. value_type is EmplaceConstructible 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 ( O(N) ) (( N ) is distance(i, j)), worst case ( O(N^2) )</td>
</tr>
<tr>
<td>X a(i, j, n, hf);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X(i, j, n)</td>
<td>X</td>
<td>Requires: hasher and key_equal are DefaultConstructible. value_type is EmplaceConstructible 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 ( O(N) ) (( N ) is distance(i, j)), worst case ( O(N^2) )</td>
</tr>
</tbody>
</table>

| § 26.2.7 | 774 |
Table 91 — 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(i, j)</code></td>
<td><code>X</code></td>
<td>Requires: hasher and key_equal are DefaultConstructible. value_type is EmplaceConstructible 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 $O(N)$ ($N$ is distance(i, j)), worst case $O(N^2)$</td>
</tr>
<tr>
<td><code>X a(i, j);</code></td>
<td><code>X</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>X(il)</code></td>
<td><code>X</code></td>
<td>Same as <code>X(il.begin(), il.end())</code>.</td>
<td>Same as <code>X(il.begin(), il.end())</code>.</td>
</tr>
<tr>
<td><code>X(il, n)</code></td>
<td><code>X</code></td>
<td>Same as <code>X(il.begin(), il.end(), n)</code>.</td>
<td>Same as <code>X(il.begin(), il.end(), n)</code>.</td>
</tr>
<tr>
<td><code>X(il, n, hf)</code></td>
<td><code>X</code></td>
<td>Same as <code>X(il.begin(), il.end(), n, hf)</code>.</td>
<td>Same as <code>X(il.begin(), il.end(), n, hf)</code>.</td>
</tr>
<tr>
<td><code>X(il, n, hf, eq)</code></td>
<td><code>X</code></td>
<td>Same as <code>X(il.begin(), il.end(), n, hf, eq)</code>.</td>
<td>Same as <code>X(il.begin(), il.end(), n, hf, eq)</code>.</td>
</tr>
<tr>
<td><code>X(b)</code></td>
<td><code>X</code></td>
<td>Copy constructor. In addition to the requirements of Table 83, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
</tr>
<tr>
<td><code>X a(b);</code></td>
<td><code>X</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a = b</code></td>
<td><code>X&amp;</code></td>
<td>Copy assignment operator. In addition to the requirements of Table 83, copies the hash function, predicate, and maximum load factor.</td>
<td>Average case linear in b.size(), worst case quadratic.</td>
</tr>
<tr>
<td><code>a = il</code></td>
<td><code>X&amp;</code></td>
<td>Requires: value_type is CopyInsertable into X and CopyAssignable. 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 <code>a = X(il)</code>.</td>
</tr>
<tr>
<td><code>b.hash_function()</code></td>
<td><code>hasher</code></td>
<td>Returns b’s hash function.</td>
<td>constant</td>
</tr>
<tr>
<td><code>b.key_eq()</code></td>
<td><code>key_equal</code></td>
<td>Returns b’s key equality predicate.</td>
<td>constant</td>
</tr>
</tbody>
</table>

§ 26.2.7
Table 91 — 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_uniq.emplace(args)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td>Requires: <code>value_type</code> shall be <code>EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Inserts a <code>value_type</code> object <code>t</code> constructed with <code>std::forward&lt;Args&gt;(args)</code>... if and only if there is no element in the container with key equivalent to the key of <code>t</code>. The <code>bool</code> component of the returned pair is <code>true</code> if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of <code>t</code>.</td>
<td>Average case (O(1)), worst case (O(a_uniq.size())).</td>
</tr>
<tr>
<td><code>a_eq.emplace(args)</code></td>
<td>iterator</td>
<td>Requires: <code>value_type</code> shall be <code>EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Inserts a <code>value_type</code> object <code>t</code> constructed with <code>std::forward&lt;Args&gt;(args)</code>... and returns the iterator pointing to the newly inserted element.</td>
<td>Average case (O(1)), worst case (O(a_eq.size())).</td>
</tr>
<tr>
<td><code>a.emplace_hint(p, args)</code></td>
<td>iterator</td>
<td>Requires: <code>value_type</code> shall be <code>EmplaceConstructible</code> into <code>X</code> from <code>args</code>. Effects: Equivalent to <code>a.emplace(std::forward&lt;Args&gt;(args))</code>. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The <code>const_iterator p</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case (O(1)), worst case (O(a.size())).</td>
</tr>
<tr>
<td><code>a_uniq.insert(t)</code></td>
<td><code>pair&lt;iterator, bool&gt;</code></td>
<td>Requires: If <code>t</code> is a non-constant value expression, <code>value_type</code> shall be <code>MoveInsertable</code> into <code>X</code>; otherwise, <code>value_type</code> shall be <code>CopyInsertable</code> into <code>X</code>. Effects: Inserts <code>t</code> if and only if there is no element in the container with key equivalent to the key of <code>t</code>. The <code>bool</code> component of the returned pair indicates whether the insertion takes place, and the <code>iterator</code> component points to the element with key equivalent to the key of <code>t</code>.</td>
<td>Average case (O(1)), worst case (O(a_uniq.size())).</td>
</tr>
</tbody>
</table>
Table 91 — 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_eq.insert(t)</code></td>
<td>iterator</td>
<td>Requires: If ( t ) is a non-const rvalue expression, ( \text{value_type} ) shall be <code>MoveInsertable</code> into ( X ); otherwise, ( \text{value_type} ) shall be <code>CopyInsertable</code> into ( X ). Effects: Inserts ( t ), and returns an iterator pointing to the newly inserted element.</td>
<td>Average case ( O(1) ), worst case ( O(a_eq.size()) ).</td>
</tr>
<tr>
<td><code>a.insert(p, t)</code></td>
<td>iterator</td>
<td>Requires: If ( t ) is a non-const rvalue expression, ( \text{value_type} ) shall be <code>MoveInsertable</code> into ( X ); otherwise, ( \text{value_type} ) shall be <code>CopyInsertable</code> into ( X ). Effects: Equivalent to <code>a.insert(t)</code>. Return value is an iterator pointing to the element with the key equivalent to that of ( t ). The iterator ( p ) is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.</td>
<td>Average case ( O(1) ), worst case ( O(a.size()) ).</td>
</tr>
<tr>
<td><code>a.insert(i, j)</code></td>
<td>void</td>
<td>Requires: ( \text{value_type} ) shall be <code>EmplaceConstructible</code> into ( X ) from ( \ast i ). Requires: ( i ) and ( j ) are not iterators in ( a ). Effects: Equivalent to <code>a.insert(t)</code> for each element in ( [i,j) ).</td>
<td>Average case ( O(N) ), where ( N ) is distance(i, j). Worst case ( O(N(a.size()) + 1) ).</td>
</tr>
<tr>
<td><code>a.insert(il)</code></td>
<td>void</td>
<td>Same as <code>a.insert(il.begin(), il.end())</code>.</td>
<td>Same as <code>a.insert(il.begin(), il.end())</code>.</td>
</tr>
</tbody>
</table>
Table 91 — 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_uniq.insert(nh)</code></td>
<td><code>insert_return_type</code></td>
<td>Requires: <code>nh</code> is empty or <code>a_uniq.get_allocator() == nh.get_allocator()</code>. Effects: If <code>nh</code> is empty, has no effect. Otherwise, inserts the element owned by <code>nh</code> if and only if there is no element in the container with a key equivalent to <code>nh.key()</code>. Postconditions: If <code>nh</code> is empty, inserted is <code>false</code>, position is <code>end()</code>, and node is empty. Otherwise if the insertion took place, inserted is <code>true</code>, position points to the inserted element, and node is empty; if the insertion failed, inserted is <code>false</code>, node has the previous value of <code>nh</code>, and position points to an element with a key equivalent to <code>nh.key()</code>.</td>
<td>Average case $O(1)$, worst case $O(a_uniq.size())$.</td>
</tr>
<tr>
<td><code>a_eq.insert(nh)</code></td>
<td><code>iterator</code></td>
<td>Requires: <code>nh</code> is empty or <code>a_eq.get_allocator() == nh.get_allocator()</code>. Effects: If <code>nh</code> is empty, has no effect and returns <code>a_eq.end()</code>. Otherwise, inserts the element owned by <code>nh</code> and returns an iterator pointing to the newly inserted element. Postconditions: <code>nh</code> is empty.</td>
<td>Average case $O(1)$, worst case $O(a_eq.size())$.</td>
</tr>
</tbody>
</table>
Table 91 — 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>Requires: 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. Postconditions: nh is empty if insertion succeeds, unchanged if insertion fails.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.extract(k)</td>
<td>node_type</td>
<td>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 $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.extract(q)</td>
<td>node_type</td>
<td>Removes the element pointed to by q. Returns a node_type owning that element.</td>
<td>Average case $\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.merge(a2)</td>
<td>void</td>
<td>Requires: a.get_allocator() == a2.get_allocator(). Attempts to extract each element in a2 and insert it into a using the hash function and key equality predicate of a. In containers with unique keys, if there is an element in a with key equivalent to the key of an element from a2, then that element is not extracted from a2. Postconditions: Pointers and references to the transferred elements of a2 refer to those same elements but as members of a. Iterators referring to the transferred elements and all iterators referring to a will be invalidated, but iterators to elements remaining in a2 will remain valid.</td>
<td>Average case $\mathcal{O}(N)$, where $N$ is a2.size(). Worst case $\mathcal{O}(N*a.size() + N)$.</td>
</tr>
</tbody>
</table>
Table 91 — 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.erase(k)</td>
<td>size_type</td>
<td>Erases all elements with key equivalent to k. Returns the number of elements erased.</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\mathcal{O}(a.count(k))$. Worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q)</td>
<td>iterator</td>
<td>Erases the element pointed to by q. Returns the iterator immediately following q prior to the erasure.</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(r)</td>
<td>iterator</td>
<td>Erases the element pointed to by r. Returns the iterator immediately following r prior to the erasure.</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\mathcal{O}(1)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.erase(q1, q2)</td>
<td>iterator</td>
<td>Erases all elements in the range [q1, q2). Returns the iterator immediately following the erased elements prior to the erasure.</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>linear in $distance(q1, q2)$, worst case $\mathcal{O}(a.size())$.</td>
</tr>
<tr>
<td>a.clear()</td>
<td>void</td>
<td>Erases all elements in the container. Postconditions: a.empty() returns true.</td>
<td>Linear in a.size().</td>
</tr>
<tr>
<td>b.find(k)</td>
<td>iterator;</td>
<td>Returns an iterator pointing to an element with key equivalent to k, or b.end() if no such element exists.</td>
<td>Average case</td>
</tr>
<tr>
<td>const_iterator for const b.</td>
<td></td>
<td></td>
<td>$\mathcal{O}(1)$, worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td>b.count(k)</td>
<td>size_type</td>
<td>Returns the number of elements with key equivalent to k.</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\mathcal{O}(b.count(k))$, worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td>b.equal_range(k)</td>
<td>pair&lt;iterator, iterator&gt;; pair&lt;const_iterator, const_iterator&gt; for const b.</td>
<td>Returns a range containing all elements with keys equivalent to k. Returns make_pair(b.end(), b.end()) if no such elements exist.</td>
<td>Average case</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\mathcal{O}(b.count(k))$. Worst case $\mathcal{O}(b.size())$.</td>
</tr>
<tr>
<td>b.bucket_count()</td>
<td>size_type</td>
<td>Returns the number of buckets that b contains.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.max_bucket_count()</td>
<td>size_type</td>
<td>Returns an upper bound on the number of buckets that b might ever contain.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.bucket(k)</td>
<td>size_type</td>
<td>Requires: b.bucket_count() &gt; 0. Returns the index of the bucket in which elements with keys equivalent to k would be found, if any such element existed. Postconditions: the return value shall be in the range [0, b.bucket_count()].</td>
<td>Constant</td>
</tr>
<tr>
<td>b.bucket_size(n)</td>
<td>size_type</td>
<td>Requires: n shall be in the range [0, b.bucket_count()). Returns the number of elements in the n\textsuperscript{th} bucket.</td>
<td>$\mathcal{O}(b.bucket_\text{-size}(n))$.</td>
</tr>
</tbody>
</table>
Table 91 — 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.begin(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td>Requires: n shall be in the range [0, b.bucket_count()). b.begin(n) returns an iterator referring to the first element in the bucket. If the bucket is empty, then b.begin(n) == b.end(n).</td>
<td>Constant</td>
</tr>
<tr>
<td>b.end(n)</td>
<td>local_iterator; const_local_iterator for const b.</td>
<td>Requires: n shall be in the range [0, b.bucket_count()). b.end(n) returns an iterator which is the past-the-end value for the bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.cbegin(n)</td>
<td>const_local_iterator</td>
<td>Requires: n shall be in the range [0, b.bucket_count()). b.cbegin(n) returns an iterator referring to the first element in the bucket. If the bucket is empty, then b.cbegin(n) == b.cend(n).</td>
<td>Constant</td>
</tr>
<tr>
<td>b.cend(n)</td>
<td>const_local_iterator</td>
<td>Requires: n shall be in the range [0, b.bucket_count()). b.cend(n) returns an iterator which is the past-the-end value for the bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.load_factor()</td>
<td>float</td>
<td>Returns the average number of elements per bucket.</td>
<td>Constant</td>
</tr>
<tr>
<td>b.max_load_factor()</td>
<td>float</td>
<td>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.</td>
<td>Constant</td>
</tr>
<tr>
<td>a.max_load_factor(z)</td>
<td>void</td>
<td>Requires: z shall be positive. May change the container’s maximum load factor, using z as a hint.</td>
<td>Constant</td>
</tr>
<tr>
<td>a.reserve(n)</td>
<td>void</td>
<td>Same as a.rehash(ceil(n / a.max_load_factor()))).</td>
<td>Average case linear in a.size(), worst case quadratic.</td>
</tr>
</tbody>
</table>

12 Two unordered containers a and b compare equal if a.size() == b.size() and, for every equivalent-key group [Ea1, Ea2) obtained from a.equal_range(Ea1), there exists an equivalent-key group [Eb1, Eb2) obtained from b.equal_range(Ea1), such that is_permutation(Ea1, Ea2, Eb1, Eb2) returns true. For unordered_set and unordered_map, the complexity of operator== (i.e., the number of calls to the ==

§ 26.2.7
operator of the `value_type`, to the predicate returned by `key_eq()`, and to the hasher returned by `hash_function()`) is proportional to \( N \) in the average case and to \( N^2 \) in the worst case, where \( N \) is `a.size()`. For `unordered_multiset` and `unordered_multimap`, the complexity of `operator==` is proportional to \( \sum E_i^2 \) in the average case and to \( N^2 \) in the worst case, where \( N \) is `a.size()`, and \( E_i \) is the size of the \( i \)th equivalent-key group in `a`. However, if the respective elements of each corresponding pair of equivalent-key groups \( E_a \) and \( E_b \) are arranged in the same order (as is commonly the case, e.g., if `a` and `b` are unmodified copies of the same container), then the average-case complexity for `unordered_multiset` and `unordered_multimap` becomes proportional to \( N \) (but worst-case complexity remains \( \Theta(N^2) \), e.g., for a pathologically bad hash function). The behavior of a program that uses `operator==` or `operator!=` on unordered containers is undefined unless the `Hash` and `Pred` function objects respectively have the same behavior for both containers and the equality comparison function for `Key` is a refinement\(^{260} \) of the partition into equivalent-key groups produced by `Pred`.

13 The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are constant iterators.

14 The `insert` and `emplace` members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The `erase` members shall invalidate only iterators and references to the erased elements, and preserve the relative order of the elements that are not erased.

15 The `insert` and `emplace` members shall not affect the validity of iterators if \((N+n) \leq z \times B\), where \( N \) is the number of elements in the container prior to the insert operation, \( n \) is the number of elements inserted, \( B \) is the container’s bucket count, and \( z \) is the container’s maximum load factor.

16 The `extract` members invalidate only iterators to the removed element, and preserve the relative order of the elements that are not erased; pointers and references to the removed element remain valid. However, accessing the element through such pointers and references while the element is owned by a `node_type` is undefined behavior. References and pointers to an element obtained while it is owned by a `node_type` are invalidated if the element is successfully inserted.

17 A deduction guide for an unordered associative container shall not participate in overload resolution if any of the following are true:

- It has an `InputIterator` template parameter and a type that does not qualify as an input iterator is deduced for that parameter.
- It has an `Allocator` template parameter and a type that does not qualify as an allocator is deduced for that parameter.
- It has a `Hash` template parameter and an integral type or a type that qualifies as an allocator is deduced for that parameter.
- It has a `Pred` template parameter and a type that qualifies as an allocator is deduced for that parameter.

26.2.7.1 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.

26.3 Sequence containers

26.3.1 In general

The headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, and `<vector>` define class templates that meet the requirements for sequence containers.

\(^{260}\) Equality comparison is a refinement of partitioning if no two objects that compare equal fall into different partitions.
26.3.2 Header <array> synopsis

```cpp
#include <initializer_list>

namespace std {
    // 26.3.7, class template array
    template<class T, size_t N> struct array;

    template<class T, size_t N>
    bool operator==(const array<T, N>& x, const array<T, N>& y);

    template<class T, size_t N>
    bool operator!=(const array<T, N>& x, const array<T, N>& y);

    template<class T, size_t N>
    bool operator< (const array<T, N>& x, const array<T, N>& y);

    template<class T, size_t N>
    bool operator> (const array<T, N>& x, const array<T, N>& y);

    template<class T, size_t N>
    bool operator<=(const array<T, N>& x, const array<T, N>& y);

    template<class T, size_t N>
    bool operator>=(const array<T, N>& x, const array<T, N>& y);

    template<class T, size_t N>
    void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));

    template<class T> class tuple_size;

    template<size_t I, class T> class 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;
}
```

26.3.3 Header <deque> synopsis

```cpp
#include <initializer_list>

namespace std {
    // 26.3.8, 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>
    void swap(deque<T, Allocator>& x, deque<T, Allocator>& y) noexcept(noexcept(x.swap(y)));
}
```
namespace pmr {
    template<class T>
    using deque = std::deque<T, polymorphic_allocator<T>>;
}

26.3.4  Header <forward_list> synopsis

#include <initializer_list>

namespace std {
    // 26.3.9, class template
    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)));

    namespace pmr {
        template<class T>
        using forward_list = std::forward_list<T, polymorphic_allocator<T>>;
    }
}

26.3.5  Header <list> synopsis

#include <initializer_list>

namespace std {
    // 26.3.10, class template
    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)));

    namespace pmr {
        template<class T>
        using list = std::list<T, polymorphic_allocator<T>>;
    }
}
26.3.6 Header <vector> synopsis

```cpp
#include <initializer_list>

namespace std {

    // 26.3.11, 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)));

    // 26.3.12, 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>>;

    namespace pmr {
        template<class T>
        using vector = std::vector<T, polymorphic_allocator<T>>;
    }
}
```

26.3.7 Class template array

26.3.7.1 Class template array overview

1 The header <array> defines a class template for storing fixed-size sequences of objects. An `array` is a contiguous container (26.2.1). 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 (11.6.1) 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 (26.2), 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 (26.2.3). 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.

```cpp
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 26.2
using const_iterator = implementation-defined; // see 26.2
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;

// no explicit construct/copy/destroy for aggregate type
void fill(const T& u);
void swap(array&) noexcept(is_nothrow_swappable_v<T>);

// iterators
constexpr iterator begin() noexcept;
constexpr const_iterator begin() const noexcept;
constexpr iterator end() noexcept;
constexpr const_iterator end() const noexcept;
constexpr reverse_iterator rbegin() noexcept;
constexpr const_reverse_iterator rbegin() const noexcept;
constexpr reverse_iterator rend() noexcept;
constexpr const_reverse_iterator rend() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
constexpr const_reverse_iterator crbegin() const noexcept;
constexpr const_reverse_iterator crend() const noexcept;

// capacity
[[nodiscard]] constexpr bool empty() const noexcept;
constexpr size_type size() const noexcept;
constexpr size_type max_size() const noexcept;

// element access
constexpr reference operator[](size_type n);
constexpr const_reference operator[](size_type n) const;
constexpr reference at(size_type n);
constexpr const_reference at(size_type n) const;
constexpr reference front();
constexpr const_reference front() const;
constexpr reference back();
constexpr const_reference back() const;

constexpr T * data() noexcept;
constexpr const T * data() const noexcept;

};

template<class T, class... U>
array(T, U...) -> array<T, 1 + sizeof...(U)>;

26.3.7.2 array constructors, copy, and assignment

1 The conditions for an aggregate (11.6.1) shall be met. Class array relies on the implicitly-declared special
member functions (15.1, 15.4, and 15.8) to conform to the container requirements table in 26.2. 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 MoveConstructible or MoveAssignable, respectively.

2 Requires: (is_same_v<T, U> && ...) is true. Otherwise the program is ill-formed.
26.3.7.3 array member functions

constexpr size_type size() const noexcept;
1
   Returns: N.
constexpr T* data() noexcept;
constexpr const T* data() const noexcept;
2
   Returns: A pointer such that data() == addressof(front()), and [data(), data() + size()) is a valid range.

void fill(const T& u);
3
   Effects: As if by fill_n(begin(), N, u).
void swap(array& y) noexcept(is_nothrow_swappable_v<T>);
4
   Effects: Equivalent to swap_ranges(begin(), end(), y.begin()).
   [ Note: Unlike the swap function for other containers, array::swap takes linear time, may exit via an exception, and does not cause iterators to become associated with the other container. — end note ]

26.3.7.4 array specialized algorithms

template<class T, size_t N>
   void swap(array<T, N>& x, array<T, N>& y) noexcept(noexcept(x.swap(y)));
1
   Remarks: This function shall not participate in overload resolution unless N == 0 or is_swappable_v<T> is true.
   Effects: As if by x.swap(y).
   Complexity: Linear in N.

26.3.7.5 Zero sized arrays

array shall provide support for the special case N == 0.
1
   In the case that N == 0, begin() == end() == unique value. The return value of data() is unspecified.
2
   The effect of calling front() or back() for a zero-sized array is undefined.
3
   Member function swap() shall have a non-throwing exception specification.

26.3.7.6 Tuple interface to class template array

template<class T, size_t N>
   struct tuple_size<array<T, N>> : integral_constant<size_t, N> { };
tuple_element<I, array<T, N>>::type
1
   Requires: I < N. The program is ill-formed if I is out of bounds.
   Value: The type T.

template<size_t I, class T, size_t N>
   constexpr T& get(array<T, N>& a) noexcept;
template<size_t I, class T, size_t N>
   constexpr T&& get(array<T, N>&& a) noexcept;
template<size_t I, class T, size_t N>
   constexpr const T& get(const array<T, N>& a) noexcept;
template<size_t I, class T, size_t N>
   constexpr const T&& get(const array<T, N>&& a) noexcept;
3
   Requires: I < N. The program is ill-formed if I is out of bounds.
   Returns: A reference to the Ith element of a, where indexing is zero-based.

26.3.8 Class template deque

26.3.8.1 Class template deque overview

A deque is a sequence container that supports random access iterators (27.2.7). In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take
linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. Storage management is handled automatically.

2 A deque satisfies all of the requirements of a container, of a reversible container (given in tables in 26.2), of a sequence container, including the optional sequence container requirements (26.2.3), and of an allocator-aware container (Table 86). Descriptions are provided here only for operations on deque that are not described in one of these tables or for operations where there is additional semantic information.

namespace std {
    template<class T, class Allocator = allocator<T>>
    class deque {
        public:
            // 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 26.2
            using difference_type = implementation-defined; // see 26.2
            using iterator = implementation-defined; // see 26.2
            using const_iterator = implementation-defined; // see 26.2
            using reverse_iterator = std::reverse_iterator<iterator>;
            using const_reverse_iterator = std::reverse_iterator<const_iterator>;

            // 26.3.8.2, construct/copy/destroy
            deque() : deque(Allocator()) { }
            explicit deque(const Allocator&);
            deque(size_type n, const Allocator&);
            template<class InputIterator>
            deque(InputIterator first, InputIterator last, const Allocator&);
            deque(const deque& x);
            deque(deque&&);
            deque(const deque&, const Allocator&);
            deque(deque&&, const Allocator&);
            deque(initializer_list<T>, const Allocator&);
            ~deque();
            deque& operator=(const deque& x);
            deque& operator=(deque&& x) noexcept;
            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;
            iterator cbegin() const noexcept;
            const_iterator cend() const noexcept;
            const_reverse_iterator crbegin() const noexcept;
            const_reverse_iterator crend() const noexcept;
        }
    }
}
// 26.3.8.3, 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;

// 26.3.8.4, 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<typename iterator_traits<InputIterator>::value_type>>
deque<InputIterator, Allocator> ::value_type>>
deque(InputIterator, InputIterator, Allocator = Allocator())
   -> deque<typename iterator_traits<InputIterator>::value_type, Allocator>;

// 26.3.8.5, specialized algorithms
template<class T, class Allocator>
void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
   noexcept(noexcept(x.swap(y)));

26.3.8.2 deque constructors, copy, and assignment
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  \textit{Effects:} Constructs a \texttt{deque} with \( n \) default-inserted elements using the specified allocator.
4  \textit{Requires:} \( T \) shall be \texttt{DefaultInsertable} into *\texttt{this}.
5  \textit{Complexity:} Linear in \( n \).

deque(size_type n, const T& value, const Allocator& = Allocator());
6  \textit{Effects:} Constructs a \texttt{deque} with \( n \) copies of \texttt{value}, using the specified allocator.
7  \textit{Requires:} \( T \) shall be \texttt{CopyInsertable} into *\texttt{this}.
8  \textit{Complexity:} Linear in \( n \).

template<class InputIterator>
deque(InputIterator first, InputIterator last, const Allocator& = Allocator());
9  \textit{Effects:} Constructs a \texttt{deque} equal to the range \([\texttt{first}, \texttt{last})\), using the specified allocator.
10 \textit{Complexity:} Linear in \texttt{distance(first, last)}.

26.3.8.3 deque capacity

void resize(size_type sz);
1  \textit{Effects:} If \( sz < \texttt{size()} \), erases the last \( \texttt{size()} - sz \) elements from the sequence. Otherwise, appends \( sz - \texttt{size()} \) default-inserted elements to the sequence.
2  \textit{Requires:} \( T \) shall be \texttt{MoveInsertable} and \texttt{DefaultInsertable} into *\texttt{this}.

void resize(size_type sz, const T& c);
3  \textit{Effects:} If \( sz < \texttt{size()} \), erases the last \( \texttt{size()} - sz \) elements from the sequence. Otherwise, appends \( sz - \texttt{size()} \) copies of \( c \) to the sequence.
4  \textit{Requires:} \( T \) shall be \texttt{CopyInsertable} into *\texttt{this}.

void shrink_to_fit();
5  \textit{Requires:} \( T \) shall be \texttt{MoveInsertable} into *\texttt{this}.
6  \textit{Effects:} \texttt{shrink_to_fit} is a non-binding request to reduce memory use but does not change the size of the sequence. [\textit{Note:} The request is non-binding to allow latitude for implementation-specific optimizations. \textit{— end note}] If an exception is thrown other than by the move constructor of a non-copy\texttt{CopyInsertable} \( T \) there are no effects.
7  \textit{Complexity:} Linear in the size of the sequence.
8  \textit{Remarks:} \texttt{shrink_to_fit} invalidates all the references, pointers, and iterators referring to the elements in the sequence as well as the past-the-end iterator.

26.3.8.4 deque modifiers

iterator insert(const_iterator position, const T& x);
1  \textit{Effects:} An insertion in the middle of the \texttt{deque} invalidates all the iterators and references to elements of the \texttt{deque}. An insertion at either end of the \texttt{deque} invalidates all the iterators to the \texttt{deque}, but has no effect on the validity of references to elements of the \texttt{deque}.
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-CopyInsertable \( T \), the effects are unspecified.

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 either at the beginning or end of a deque always takes constant time and causes a single call to a constructor of \( T \).

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
void pop_front();
void pop_back();
```

Effects: An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only iterators and references to the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque. [ Note: \texttt{pop\_front} and \texttt{pop\_back} are erase operations. — end note ]

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.

Throws: Nothing unless an exception is thrown by the assignment operator of \( T \).

### 26.3.8.5 deque specialized algorithms

```
template<class T, class Allocator>
void swap(deque<T, Allocator>& x, deque<T, Allocator>& y)
   noexcept(noexcept(x.swap(y)));
```

Effects: As if by \texttt{x.swap(y)}.

### 26.3.9 Class template forward_list

#### 26.3.9.1 Class template forward_list overview

A \texttt{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 \texttt{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 \texttt{forward\_list} satisfies all of the requirements of a container (Table 83), except that the \texttt{size()} member function is not provided and \texttt{operator\=} has linear complexity. A \texttt{forward\_list} also satisfies all of the requirements for an allocator-aware container (Table 86). In addition, a \texttt{forward\_list} provides the \texttt{assign} member functions (Table 87) and several of the optional container requirements (Table 88). Descriptions are provided here only for operations on \texttt{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 \texttt{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 \texttt{erase} and \texttt{splice}, must be open at the beginning. — end note ]

```
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 26.2
```
using difference_type = implementation-defined; // see 26.2
using iterator = implementation-defined; // see 26.2
using const_iterator = implementation-defined; // see 26.2

// 26.3.9.2, construct/copy/destroy
forward_list() : forward_list(Allocator()) { }
explicit forward_list(const Allocator&);
explicit forward_list(size_type n, const Allocator& = Allocator());
forward_list(const_iterator value, const Allocator& = Allocator());
template<class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
forward_list(const_iterator x);
forward_list(const_iterator 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);
forward_list& operator=(initializer_list<T>, const Allocator& = Allocator());

// 26.3.9.3, iterators
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cbefore_begin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type max_size() const noexcept;

// 26.3.9.4, element access
reference front();
const_reference front() const;

// 26.3.9.5, 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;

// 26.3.9.6, 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);

void remove(const T& value);
template<class Predicate> void remove_if(Predicate pred);

void unique();
template<class BinaryPredicate> void unique(BinaryPredicate binary_pred);

void merge(forward_list& x);
void merge(forward_list&& x);
template<class Compare> void merge(forward_list& x, Compare comp);
template<class Compare> void merge(forward_list&& x, Compare comp);

void sort();
template<class Compare> void sort(Compare comp);

void reverse() noexcept;
};

template<class InputIterator,
   class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
forward_list(InputIterator, InputIterator, Allocator = Allocator())
-> forward_list<typename iterator_traits<InputIterator>::value_type, Allocator>;

// 26.3.9.7, specialized algorithms
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 (20.5.3.5.1). T shall be complete before any member of the resulting specialization of forward_list is referenced.

26.3.9.2 forward_list constructors, copy, 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   Effects: Constructs a forward_list object with n default-inserted elements using the specified allocator.
4   Requires: T shall be DefaultInsertable into *this.
5   Complexity: Linear in n.

forward_list(size_type n, const T& value, const Allocator& = Allocator());
6   Effects: Constructs a forward_list object with n copies of value using the specified allocator.

§ 26.3.9.2
Requires: \(T\) shall be CopyInsertable into \(*\text{this}.*\)

Complexity: Linear in \(n\).

```cpp
template<class InputIterator>
forward_list(InputIterator first, InputIterator last, const Allocator& = Allocator());
```

Effects: Constructs a `forward_list` object equal to the range \([\text{first}, \text{last})\).

Complexity: Linear in \(\text{distance(}\text{first}, \text{last})\).

26.3.9.3 `forward_list` iterators

```cpp
iterator before_begin() noexcept;
const_iterator before_begin() const noexcept;
const_iterator cbefore_begin() const noexcept;
```

Returns: A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by `begin()`.

Effects: `cbefore_begin()` is equivalent to `const_cast<forward_list const&>(*this).before_begin()`.

Remarks: \(\text{before_begin()} == \text{end()}\) shall equal false.

26.3.9.4 `forward_list` element access

```cpp
reference front();
const_reference front() const;
```

Returns: \(*\text{begin}()\).

26.3.9.5 `forward_list` modifiers

None of the overloads of `insert_after` shall affect the validity of iterators and references, and `erase_after` shall invalidate only iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Inserting \(n\) elements into a `forward_list` is linear in \(n\), and the number of calls to the copy or move constructor of \(T\) is exactly equal to \(n\). Erasing \(n\) elements from a `forward_list` is linear in \(n\) and the number of calls to the destructor of type \(T\) is exactly equal to \(n\).

```cpp
template<class... Args> reference emplace_front(Args&&... args);
```

Effects: Inserts an object of type `value_type` constructed with `value_type(std::forward<Args>(args)...)` at the beginning of the list.

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())`.

```cpp
iterator insert_after(const_iterator position, const T& x);
it iterator insert_after(const_iterator position, T&& x);
```

Requires: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

Effects: Inserts a copy of \(x\) after `position`.

Returns: An iterator pointing to the copy of \(x\).

```cpp
iterator insert_after(const_iterator position, size_type n, const T& x);
```

Requires: `position` is `before_begin()` or is a dereferenceable iterator in the range `[begin(), end())`.

Effects: Inserts \(n\) copies of \(x\) after `position`.

Returns: An iterator pointing to the last inserted copy of \(x\) or `position` if \(n == 0\).
template<class InputIterator>
iterator insert_after(const_iterator position, InputIterator first, InputIterator last);

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
first and last are not iterators in *this.

Effects: Inserts copies of elements in [first, last) after position.

Returns: An iterator pointing to the last inserted element or position if first == last.

iterator insert_after(const_iterator position, initializer_list<T> il);

Effects: insert_after(p, il.begin(), il.end()).

Returns: An iterator pointing to the last inserted element or position if il is empty.

template<class... Args>
iterator emplace_after(const_iterator position, Args&&... args);

Requires: 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);

Requires: The iterator following position is dereferenceable.

Effects: Erases the element pointed to by the iterator following position.

Returns: An iterator pointing to the element following the one that was erased, or end() if no such element exists.

Throws: Nothing.

iterator erase_after(const_iterator position, const_iterator last);

Requires: 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);

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.

Requires: T shall be DefaultInsertable into *this.

void resize(size_type sz, const value_type& c);

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.

Requires: T shall be CopyInsertable into *this.

void clear() noexcept;

Effects: Erases all elements in the range [begin(), end()).

Remarks: Does not invalidate past-the-end iterators.

26.3.9.6 forward_list operations

In this subclause, arguments for a template parameter named Predicate or BinaryPredicate shall meet the corresponding requirements in 28.3. For merge and sort, the definitions and requirements in 28.7 apply.

void splice_after(const_iterator position, forward_list&amp; x);
void splice_after(const_iterator position, forward_list&& x);

Requires: position is before_begin() or is a dereferenceable iterator in the range [begin(), end()).
get_allocator() == x.get_allocator() & x != this.

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: \( O(\text{distance}(x.\text{begin}(), x.\text{end}())) \)

void splice_after(const_iterator position, forward_list& x, const_iterator i);
void splice_after(const_iterator position, forward_list&& x, const_iterator i);

Requires: 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().

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);

Requires: 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().

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: \( O(\text{distance}((\text{first, last})) \)

void remove(const T& value);
template<class Predicate> void remove_if(Predicate pred);

Effects: Erases all the elements in the list referred 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.

 Throws: Nothing unless an exception is thrown by the equality comparison or the predicate.

Remarks: Stable (20.5.5.7).

Complexity: Exactly distance(begin(), end()) applications of the corresponding predicate.

void unique();
template<class BinaryPredicate> void 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.

 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);
20 \textit{Requires:} \texttt{*this} and \texttt{x} are both sorted with respect to the comparator \texttt{operator<} (for the first two overloads) or \texttt{comp} (for the last two overloads), and \texttt{get\_allocator()} == \texttt{x.get\_allocator()} is true.

21 \textit{Effects:} Merges the two sorted ranges \(\texttt{[begin(), end())}\) and \(\texttt{[x.begin(), x.end())}\). \texttt{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 \texttt{x} now refer to those same elements but as members of \texttt{*this}. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \texttt{*this}, not into \texttt{x}.

22 \textit{Remarks:} Stable (20.5.5.7). The behavior is undefined if \texttt{get\_allocator()} != \texttt{x.get\_allocator()}.

23 \textit{Complexity:} At most \(\texttt{distance(begin(), end()) + distance(x.begin(), x.end()) - 1}\) comparisons.

24 \textit{Effects:} Sorts the list according to the \texttt{operator<} or the \texttt{comp} function object. If an exception is thrown, the order of the elements in \texttt{*this} is unspecified. Does not affect the validity of iterators and references.

25 \textit{Remarks:} Stable (20.5.5.7).

26 \textit{Complexity:} Approximately \(N \log N\) comparisons, where \(N\) is \texttt{distance(begin(), end())}.

27 \textit{Effects:} Reverses the order of the elements in the list. Does not affect the validity of iterators and references.

28 \textit{Complexity:} Linear time.

26.3.9.7 \textit{forward\_list} specialized algorithms \hfil [forwardlist.spec]

26.3.10 \textit{Class template list} \hfil [list]

26.3.10.1 \textit{Class template list overview} \hfil [list.overview]

1 A \texttt{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 (26.3.11) and deques (26.3.8), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

2 A \texttt{list} satisfies all of the requirements of a container, of a reversible container (given in two tables in 26.2), of a sequence container, including most of the optional sequence container requirements (26.2.3), and of an allocator-aware container (Table 86). The exceptions are the \texttt{operator[]} and \texttt{at} member functions, which are not provided.\footnote{These member functions are only provided by containers whose iterators are random access iterators.} Descriptions are provided here only for operations on \texttt{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 26.2
using difference_type = implementation-defined; // see 26.2
using iterator = implementation-defined; // see 26.2
using const_iterator = implementation-defined; // see 26.2
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;

// 26.3.10.2, construct/copy/destroy
list() : list(Allocator()) { }
explicit list(const Allocator&);
explicit list(size_type n, const Allocator& = Allocator());
list(size_type n, const T& value, const Allocator& = Allocator());
template<class InputIterator>
    list(InputIterator first, InputIterator last, const Allocator& = Allocator());
list(const list& x);
list(list&& x);
list(const list&, const Allocator&);
list(list&, const Allocator&);
list(initializer_list<T>, const Allocator& = Allocator());
~list();
list& operator=(const list& x);
list& operator=(list&& x) noexcept(allocator_traits<Allocator>::is_always_equal::value);
list& operator=(initializer_list<T>);
template<class InputIterator>
    void assign(InputIterator first, InputIterator last);
void assign(size_type n, const T& t);
void assign(initializer_list<T>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
reverse_iterator rbegin() noexcept;
const_reverse_iterator rbegin() const noexcept;
reverse_iterator rend() noexcept;
const_reverse_iterator rend() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 26.3.10.3, 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;

// 26.3.10.4, modifiers
template<class... Args> reference emplace_front(Args&&... args); 
template<class... Args> reference emplace_back(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();
void push_back(const T& x);
void push_back(T&& x);
void pop_back();

template<class... Args> iterator emplace(const_iterator position, Args&&... args);
iterator insert(const_iterator position, const T& x);
iterator insert(const_iterator position, T&& x);
iterator insert(const_iterator position, size_type n, const T& x);
template<class InputIterator>
iterator insert(const_iterator position, InputIterator first, InputIterator last);
iterator insert(const_iterator position, const_iterator first, const_iterator last);

iterator erase(const_iterator position);
iterator erase(const_iterator position, const_iterator last);

// 26.3.10.5, 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);

void remove(const T& value);
template<class Predicate> void remove_if(Predicate pred);

void unique();
template<class BinaryPredicate>
void 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<typename iterator_traits<InputIterator>::value_type>>
list(InputIterator, InputIterator, Allocator = Allocator())
-> list<typename iterator_traits<InputIterator>::value_type, Allocator>;

// 26.3.10.6, specialized algorithms
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 (20.5.3.5.1). T shall be complete before any member of the resulting specialization of list is referenced.

26.3.10.2 list constructors, copy, and assignment [list.cons]

explicit list(const Allocator&);

Effects: Constructs an empty list, using the specified allocator.

Complexity: Constant.
explicit list(size_type n, const Allocator& = Allocator());

Effects: Constructs a list with n default-inserted elements using the specified allocator.
Requires: T shall be DefaultInsertable into *this.
Complexity: Linear in n.

list(size_type n, const T& value, const Allocator& = Allocator());

Effects: Constructs a list with n copies of value, using the specified allocator.
Requires: T shall be CopyInsertable into *this.
Complexity: Linear in n.

template<class InputIterator>
list(InputIterator first, InputIterator last, const Allocator& = Allocator());

Effects: Constructs a list equal to the range [first, last).
Complexity: Linear in distance(first, last).

26.3.10.3 list capacity

void resize(size_type sz);

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());

Requires: T shall be DefaultInsertable into *this.

void resize(size_type sz, const T& c);

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

Requires: T shall be CopyInsertable into *this.

26.3.10.4 list 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.

```cpp
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);
```

**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.

### 26.3.10.5 list operations

Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.\(^{262}\) In this subclause, arguments for a template parameter named `Predicate` or `BinaryPredicate` shall meet the corresponding requirements in 28.3. For `merge` and `sort`, the definitions and requirements in 28.7 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()`.

```cpp
void splice(const_iterator position, list& x);
void splice(const_iterator position, list&& x);
```

**Requires:** \( \&x \neq \text{this} \).

**Effects:** Inserts the contents of \( x \) before \( \text{position} \) and \( x \) becomes empty. Pointers and references to the moved elements of \( x \) now refer to those same elements but as members of \( \text{this} \). Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into \( \text{this} \), not into \( x \).

**Throws:** Nothing.

**Complexity:** Constant time.

```cpp
void splice(const_iterator position, list& x, const_iterator i);
void splice(const_iterator position, list&& x, const_iterator i);
```

**Requires:** \( i \) is a valid dereferenceable iterator of \( x \).

**Effects:** Inserts an element pointed to by \( i \) from list \( x \) before \( \text{position} \) and removes the element from \( x \). The result is unchanged if \( \text{position} == i \) or \( \text{position} == ++i \). Pointers and references to \( \*i \) continue to refer to this same element but as a member of \( \text{this} \). Iterators to \( \*i \) (including \( i \) itself) continue to refer to the same element, but now behave as iterators into \( \text{this} \), not into \( x \).

**Throws:** Nothing.

**Complexity:** Constant time.

```cpp
void splice(const_iterator position, list& x, const_iterator first, const_iterator last);
void splice(const_iterator position, list&& x, const_iterator first, const_iterator last);
```

**Requires:** \( [\text{first}, \text{last}) \) is a valid range in \( x \). The program has undefined behavior if \( \text{position} \) is an iterator in the range \( [\text{first}, \text{last}) \).

**Effects:** Inserts elements in the range \( [\text{first}, \text{last}) \) before \( \text{position} \) and removes the elements from \( x \). Pointers and references to the moved elements of \( x \) now refer to those same elements but as members

---

\(^{262}\) As specified in 20.5.3.5, the requirements in this Clause apply only to lists whose allocators compare equal.
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 &x == this; otherwise, linear time.

void remove(const T& value);

Effects: Erases all the elements in the list referred 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.

Throws: Nothing unless an exception is thrown by *i == value or pred(*i) != false.

Remarks: Stable (20.5.5.7).

Complexity: Exactly size() applications of the corresponding predicate.

void unique();

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.

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);
void merge(list&& x);

Requires: 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).

Effects: If (&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 (20.5.5.7). If (&x != this) the range [x.begin(), x.end()) is empty after the merge. No elements are copied by this operation. The behavior is undefined if get_allocator() != x.get_allocator().

Complexity: At most size() + x.size() - 1 applications of comp if (&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 (20.5.5.7).
Complexity: Approximately $N \log N$ comparisons, where $N = \text{size()}$.

26.3.10.6 List specialized algorithms

```cpp
template<class T, class Allocator>
void swap(list<T, Allocator>& x, list<T, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
```

Effects: As if by x.swap(y).

26.3.11 Class template vector

26.3.11.1 Class template 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 26.2), of a sequence container, including most of the optional sequence container requirements (26.2.3), of an allocator-aware container (Table 86), and, for an element type other than `bool`, of a contiguous container (26.2.1). 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.

```cpp
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 26.2
    using difference_type = implementation-defined; // see 26.2
    using iterator = implementation-defined; // see 26.2
    using const_iterator = implementation-defined; // see 26.2
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // 26.3.11.2, construct/copy/destroy
    vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
    explicit vector(const Allocator&) noexcept;
    explicit vector(size_type n, const Allocator& = Allocator());
    vector(size_type n, const T& value, const Allocator& = Allocator());
    template<class InputIterator>
    vector(InputIterator first, InputIterator last, const Allocator& = Allocator());
    vector(const vector& x);
    vector(vector&&) noexcept);
    vector(const vector& x);
    vector(vector&& x);
    vector vector& operator=(const vector& x);
    vector vector& operator=(vector&& x)
        noexcept(allocator_traits<Allocator>::propagate_on_container_move_assignment::value ||
        allocator_traits<Allocator>::is_always_equal::value);
    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;
    ```

§ 26.3.11.1
// 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;

// 26.3.11.3, 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;

// 26.3.11.4, data access
T* data() noexcept;
const T* data() const noexcept;

// 26.3.11.5, 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);
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 first, const_iterator last);
void swap(vector&);

// element access

§ 26.3.11.1 804
template<class T, class Allocator>
void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
   noexcept(noexcept(x.swap(y)));

An incomplete type T may be used when instantiating vector if the allocator satisfies the allocator completeness requirements (20.5.3.5.1). T shall be complete before any member of the resulting specialization of vector is referenced.

26.3.11.2 vector constructors, copy, and assignment [vector.cons]

explicit vector(const Allocator&);

Effects: Constructs an empty vector, using the specified allocator.

Complexity: Constant.

explicit vector(size_type n, const Allocator& = Allocator());

Effects: Constructs a vector with n default-inserted elements using the specified allocator.

Requires: T shall be DefaultInsertable into *this.

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.

Requires: T shall be CopyInsertable into *this.

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.

26.3.11.3 vector capacity [vector.capacity]

size_type capacity() const noexcept;

Returns: The total number of elements that the vector can hold without requiring reallocation.

void reserve(size_type n);

Requires: T shall be MoveInsertable 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-CopyInsertable 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. No reallocation shall take place during insertions that happen after a call to reserve() until the time when an insertion would make the size of the vector greater than the value of capacity().

reserve() uses Allocator::allocate() which may throw an appropriate exception.

§ 26.3.11.3
void shrink_to_fit();

Requires: T shall be MoveInsertable into *this.

Effects: shrink_to_fit is a non-binding request to reduce capacity() to size(). [Note: The request is non-binding to allow latitude for implementation-specific optimizations. — end note] It does not increase capacity(), but may reduce capacity() by causing reallocation. If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.

Complexity: 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. If no reallocation happens, they remain valid.

void swap(vector& x)
    noexcept(allocator_traits<Allocator>::propagate_on_container_swap::value ||
    allocator_traits<Allocator>::is_always_equal::value);

Effects: Exchanges the contents and capacity() of *this with that of x.

Complexity: Constant time.

void resize(size_type sz);

Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() default-inserted elements to the sequence.

Requires: T shall be MoveInsertable and DefaultInsertable into *this.

Remarks: If an exception is thrown other than by the move constructor of a non-CopyInsertable T there are no effects.

void resize(size_type sz, const T& c);

Effects: If sz < size(), erases the last size() - sz elements from the sequence. Otherwise, appends sz - size() copies of c to the sequence.

Requires: T shall be CopyInsertable into *this.

Remarks: If an exception is thrown there are no effects.

26.3.11.4 Vector 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.

26.3.11.5 Vector 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);

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. If no reallocation happens, all the iterators and references before the insertion point remain valid. 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 `CopyInsertable` 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-`CopyInsertable` \( T \), the effects are unspecified.

**Complexity:** The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

```cpp
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 \( T \).

### 26.3.11.6 vector specialized algorithms

```cpp
template<class T, class Allocator>
void swap(vector<T, Allocator>& x, vector<T, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
```

**Effects:** As if by \( x.\text{swap}(y) \).

### 26.3.12 Class vector<bool>

To optimize space allocation, a specialization of vector for bool elements is provided:

```cpp
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();
            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
global
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 (23.10.9.2) 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;

5 Effects: Replaces each element in the container with its complement.

static void swap(reference x, reference y) noexcept;

6 Effects: Exchanges the contents of x and y as if by:

   bool b = x;
   x = y;
   y = b;

template<class Allocator> struct hash<vector<bool, Allocator>>;

7 The specialization is enabled (23.14.15).

26.4 Associative containers

26.4.1 In general

The header <map> defines the class templates map and multimap; the header <set> defines the class templates set and multiset.

2 The following exposition-only alias templates may appear in deduction guides for associative containers:

   template<class InputIterator>
   using iter_key_t = remove_const_t<typename iterator_traits<InputIterator>::value_type::first_type>; // exposition only

   template<class InputIterator>
   using iter_val_t = typename iterator_traits<InputIterator>::value_type::second_type; // exposition only

   template<class InputIterator>
   using iter_to_alloc_t = pair<add_const_t<typename iterator_traits<InputIterator>::value_type::first_type>,
   typename iterator_traits<InputIterator>::value_type::second_type>; // exposition only

26.4.2 Header <map> synopsis

#include <initializer_list>

namespace std {

   // 26.4.4, 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);

§ 26.4.2 809
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)));

// 26.4.5, 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)));

namespace pmr {
    template<class Key, class T, class Compare = less<Key>>
    using map = std::map<Key, T, Compare,
    polymorphic_allocator<pair<const Key, T>>;

    template<class Key, class T, class Compare = less<Key>>
    using multimap = std::multimap<Key, T, Compare,
    polymorphic_allocator<pair<const Key, T>>;
}

26.4.3 Header <set> synopsis
[associative.set.syn]
#include <initializer_list>

namespace std {
    // 26.4.6, class template set
    template<class Key, class Compare = less<Key>, class Allocator = allocator<Key>>
    class set;

§ 26.4.3 810
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)));

// 26.4.7, 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)));

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>>;
}

§ 26.4.3
26.4.4 Class template map

26.4.4.1 Class template map overview

1 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.

2 A map satisfies all of the requirements of a container, of a reversible container (26.2), of an associative container (26.2.6), and of an allocator-aware container (Table 86). A map also provides most operations described in 26.2.6 for unique keys. This means that a map supports the a_uniq operations in 26.2.6 but not the a_eq operations. For a map\(\langle Key, T\rangle\) 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 26.2
            using difference_type = implementation_defined; // see 26.2
            using iterator = implementation_defined; // see 26.2
            using const_iterator = implementation_defined; // see 26.2
            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);
                }
    };

    // 26.4.4.2, 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());

§ 26.4.4.1
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;

// 26.4.4.3, element access
T& operator[](const key_type& x);
T& operator[](key_type&& x);
T& at(const key_type& x);

// 26.4.4.4, modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& x);
pair<iterator, bool> insert(value_type&& x);

template<class P>
    iterator insert(const_iterator position, P&& x);

template<class InputIterator>
    void insert(InputIterator first, InputIterator last);

node_type extract(const_iterator position);

node_type extract(const key_type& x);

insert_return_type insert(node_type&& nh);

iterator insert(const_iterator hint, node_type&& nh);

template<class... Args>
    pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
    pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
  iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);

template<class... Args>
  iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);

template<class M>
  pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);

template<class M>
  pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);

template<class M>
  iterator insert_or_assign(const_iterator hint, const key_type& k, M&& obj);

template<class M>
  iterator insert_or_assign(const_iterator hint, key_type&& k, M&& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);

void swap(map&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
  is_nothrow_swappable_v<Compare>);

void clear() noexcept;

template<class C2>
  void merge(map<Key, T, C2, Allocator>& source);

template<class C2>
  void merge(map<Key, T, C2, Allocator>&& source);

template<class C2>
  void merge(multimap<Key, T, C2, Allocator>& source);

template<class C2>
  void merge(multimap<Key, T, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// map operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;

template<class K> iterator find(const K& x);

template<class K> const_iterator find(const K& x) const;

size_type count(const key_type& x) const;

template<class K> size_type count(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

template<class K> iterator lower_bound(const K& x);

template<class K> const_iterator lower_bound(const K& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;

template<class K> iterator upper_bound(const K& x);

template<class K> const_iterator upper_bound(const K& x) const;

pair<iterator, iterator> equal_range(const key_type& x);

pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

template<class K> pair<iterator, iterator> equal_range(const K& x);

template<class K> pair<const_iterator, const_iterator> equal_range(const K& x) const;
template<class InputIterator, class Compare = less<iter_key_t<InputIterator>>, class Allocator = allocator<iter_to_alloc_t<InputIterator>>> map(InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator())
    -> map<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare, Allocator>;

template<class Key, class T, class Compare = less<Key>, class Allocator = allocator<pair<const Key, T>>>
map(initializer_list<pair<const Key, T>>, Compare = Compare(), Allocator = Allocator())
    -> map<Key, T, Compare, Allocator>;

template<class InputIterator, class Allocator>>
map(InputIterator, InputIterator, Allocator)
    -> map<iter_key_t<InputIterator>, iter_val_t<InputIterator>, less<iter_key_t<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
map(initializer_list<pair<const Key, T>>, Allocator) -> map<Key, T, less<Key>, Allocator>;

// 26.4.4.5, specialized algorithms
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)));

26.4.4.2 map constructors, copy, and assignment [map.cons]

explicit map(const Compare& comp, const Allocator& = Allocator());

Effects: Constructs an empty map using the specified comparison object and allocator.
Complexity: Constant.

template<class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare(), const Allocator& = Allocator());

Effects: Constructs an empty map using the specified comparison object and allocator, and inserts
elements from the range [first, last).
Complexity: Linear in $N$ if the range [first, last) is already sorted using $\text{comp}$ and otherwise
$N \log N$, where $N$ is last - first.

26.4.4.3 map element access [map.access]

T& operator[](const key_type& x);
Effects: Equivalent to: return try_emplace(x).first->second;

T& operator[](key_type&& x);
Effects: Equivalent to: return try_emplace(move(x)).first->second;

T& at(const key_type& x);
const T& 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.

26.4.4.4 map modifiers [map.modifiers]

template<class P>
pair<iterator, bool> insert(P&& x);
template<class P>
iterator insert(const_iterator position, P&& x);

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)).
Remarks: These signatures shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is true.

```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);
```

Requires: `value_type` shall be `EmplaceConstructible` 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);
```

Requires: `value_type` shall be `EmplaceConstructible` 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);
```

Requires: `is_assignable_v<mapped_type&, M&&>` shall be `true`. `value_type` shall be `EmplaceConstructible` 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);
```

Requires: `is_assignable_v<mapped_type&, M&&>` shall be `true`. `value_type` shall be `EmplaceConstructible` 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.
26.4.4.5 map specialized algorithms

```cpp
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)));
```

1 Effects: As if by `x.swap(y)`.

26.4.5 Class template multimap

26.4.5.1 Class template multimap 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 (26.2), of an associative container (26.2.6), and of an allocator-aware container (Table 86). A `multimap` also provides most operations described in 26.2.6 for equal keys. This means that a `multimap` supports the `a_eq` operations in 26.2.6 but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key, T>`. Descriptions are provided here only for operations on `multimap` that are not described in one of those tables or for operations where there is additional semantic information.

```cpp
namespace std {
    template<class Key, class T, class Compare = less<Key>,
             class Allocator = allocator<pair<const Key, T>>>
    class multimap {
    public:
        // types
        using key_type = Key;
        using mapped_type = T;
        using value_type = pair<const Key, T>;
        using key_compare = Compare;
        using allocator_type = Allocator;
        using pointer = typename allocator_traits<Allocator>::pointer;
        using const_pointer = typename allocator_traits<Allocator>::const_pointer;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = implementation-defined; // see 26.2
        using difference_type = implementation-defined; // see 26.2
        using iterator = unspecified;
        using const_iterator = unspecified;
        using reverse_iterator = unspecified;
        using node_type;

        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);
                }
        };

        // 26.4.5.2, 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);
    };
}

§ 26.4.5.1

817

© ISO/IEC N4713
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());

multimap(InputIterator first, InputIterator last, const Allocator& a) { }
multimap(initializer_list<value_type> il, const Allocator& a) { }
~multimap();
multimap& operator=(const multimap& x);
multimap& operator=(multimap&& x)
    noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_move_assignable_v<Compare>);
multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
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;

// 26.4.5.3, 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);

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);

§ 26.4.5.1
void swap(multimap&) noexcept(allocator_traits<Allocator>::is_always_equal::value && is_nothrow_swappable_v<Compare>);

void clear() noexcept;

template<class C2>
void merge(multimap<Key, T, C2, Allocator>& source);
template<class C2>
void merge(multimap<Key, T, C2, Allocator>&& source);
template<class C2>
void merge(map<Key, T, C2, Allocator>& source);
template<class C2>
void merge(map<Key, T, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// map operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;

size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
template<class K> iterator lower_bound(const K& x);
template<class K> const_iterator lower_bound(const K& x) const;

iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
template<class K> iterator upper_bound(const K& x);
template<class K> const_iterator upper_bound(const K& x) const;

pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
template<class K>
pair<iterator, iterator> equal_range(const K& x);
template<class K>
pair<const_iterator, const_iterator> equal_range(const K& x) const;

};

template<class InputIterator, class Compare = less<iter_key_t<InputIterator>>, class Allocator = allocator<iter_to_alloc_t<InputIterator>>>
multimap<InputIterator, InputIterator, Compare = Compare(), Allocator = Allocator>()
-> multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Compare, Allocator>;

template<class Key, class T, class Compare = less<Key>, class Allocator = allocator<pair<const Key, T>>>
multimap(initializer_list<pair<const Key, T>>, Compare = Compare(), Allocator = Allocator())
-> multimap<Key, T, Compare, Allocator>;

template<class InputIterator, class Allocator>
multimap<InputIterator, InputIterator, Allocator>()
-> multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>, less<iter_key_t<InputIterator>>, Allocator>;

 template<class Key, class T, class Allocator>
multimap(initializer_list<pair<const Key, T>>, Allocator)
-> multimap<Key, T, less<Key>, Allocator>;

§ 26.4.5.1
26.4.5.2 multimap constructors

```cpp
explicit multimap(const Compare& comp, const Allocator& = Allocator());
```

1 Effects: Constructs an empty multimap using the specified comparison object and allocator.
2 Complexity: Constant.

```cpp
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 last - first.

26.4.5.3 multimap modifiers

```cpp
template<class P> iterator insert(P&& x);
template<class P> iterator insert(const_iterator position, P&& x);
```

1 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)).
2 Remarks: These signatures shall not participate in overload resolution unless \( \text{is_constructible}_{-v<\text{value_type}, \text{P}&&} \) is true.

26.4.5.4 multimap specialized algorithms

```cpp
void swap(multimap<Key, T, Compare, Allocator>& x,
multimap<Key, T, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));
```

1 Effects: As if by \( x\.swap(y) \).

26.4.6 Class template set

26.4.6.1 Class template set overview

A set is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. The set class supports bidirectional iterators.

A set satisfies all of the requirements of a container, of a reversible container (26.2), of an associative container (26.2.6), and of an allocator-aware container (Table 86). A set also provides most operations described in 26.2.6 for unique keys. This means that a set supports the \( \text{auniq} \) operations in 26.2.6 but not the \( \text{aeq} \) operations. For a set type \( \text{set<>} \) both the key_type and value_type are Key. Descriptions are provided here only for operations on set that are not described in one of these tables and for operations where there is additional semantic information.

```cpp
namespace std {

    template<class Key, class Compare = less<Key>,
             class Allocator = allocator<Key>>
    class set {

        public:

            using key_type = Key;
            using key_compare = Compare;
            using value_type = Key;
            using value_compare = Compare;

```
using allocator_type = Allocator;
using pointer = typename allocator_traits<Allocator>::pointer;
using const_pointer = typename allocator_traits<Allocator>::const_pointer;
using reference = value_type&;
using const_reference = const value_type&;
using size_type = implementation-defined; // see 26.2
using difference_type = implementation-defined; // see 26.2
using iterator = implementation-defined; // see 26.2
using const_iterator = implementation-defined; // see 26.2
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>;

// 26.4.6.2, construct/copy/destroy
set() : set(Compare()) {} 
explicit set(const Compare& comp, const Allocator& = Allocator());
template<class InputIterator>
  set(InputIterator first, InputIterator last,
      const Compare& comp = Compare(), const Allocator& = Allocator());
set(const set& x);
set(set&& x);
explicit set(const Allocator&);
set(const set&, const Allocator&);
set(set&, const Allocator&);
set(initializer_list<value_type>, const Compare& = Compare(),
     const Allocator& = Allocator());
template<class InputIterator>
  set(InputIterator first, InputIterator last, const Allocator& a)
    : set(first, last, Compare(), a) {} 
set(initializer_list<value_type> il, const Allocator& a)
  : set(il, Compare(), a) {}
  ~set();
set& operator=(const set& x);
set& operator=(set&& x)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
             is_nothrow_move_assignable_v<Compare>);
set& 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;

// 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;

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;
P
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 iterator_traits<InputIterator>::value_type>,  
class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
set(InputIterator, InputIterator,  
    Compare = Compare(), Allocator = Allocator())
-> set<typename iterator_traits<InputIterator>::value_type, 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>;

// 26.4.6.3, specialized algorithms
template<class Key, class Compare, class Allocator>
void swap(set<Key, Compare, Allocator>& x,  
    set<Key, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));

26.4.6.2 set 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 \([\text{first}, \text{last})\) is already sorted using \( \text{comp} \) and otherwise \( N \log N \), where \( N = \text{last} - \text{first} \).

26.4.6.3 set specialized algorithms

template<class Key, class Compare, class Allocator>
void swap(set<Key, Compare, Allocator>& x,  
    set<Key, Compare, Allocator>& y)
    noexcept(noexcept(x.swap(y)));
Effects: As if by \( x.\text{swap}(y) \).

26.4.7 Class template multiset

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 (26.2), of an associative container (26.2.6), and of an allocator-aware container (Table 86). multiset also provides most operations described in 26.2.6 for duplicate keys. This means that a multiset supports the \( \text{a_eq} \) operations in 26.2.6 but not the \( \text{a_uniq} \) operations. For a multiset\( <\text{Key}> \) both the \text{key_type} and \text{value_type} are \text{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 26.2
        using difference_type = implementation-defined; // see 26.2
        using iterator = implementation-defined; // see 26.2
        using const_iterator = implementation-defined; // see 26.2
        using reverse_iterator = std::reverse_iterator<iterator>;
        using const_reverse_iterator = std::reverse_iterator<const_iterator>;
        using node_type = unspecified;

        // 26.4.7.2, 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(const Allocator&, 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>);
        allocator_type get_allocator() const noexcept;

        // iterators
        iterator begin() noexcept;
        const_iterator begin() const noexcept;
        iterator end() noexcept;
        const_iterator end() const noexcept;
        reverse_iterator rbegin() noexcept;
        const_reverse_iterator rbegin() const noexcept;
        reverse_iterator rend() noexcept;
        const_reverse_iterator rend() const noexcept;
        const_iterator cbegin() const noexcept;
        const_iterator cend() const noexcept;
        const_reverse_iterator crbegin() const noexcept;
        const_reverse_iterator crend() const noexcept;
    }
// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& x);
iterator insert(value_type&& x);
iterator insert(const_iterator position, const value_type& x);
iterator insert(const_iterator position, value_type&& x);
template<class InputIterator>
  void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& x);
iterator erase(const_iterator first, const_iterator last);
void swap(multiset&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
  is_nothrow_swappable_v<Compare>);
void clear() noexcept;

template<class C2>
  void merge(multiset<Key, C2, Allocator>& source);
template<class C2>
  void merge(multiset<Key, C2, Allocator>&& source);
template<class C2>
  void merge(set<Key, C2, Allocator>& source);
template<class C2>
  void merge(set<Key, C2, Allocator>&& source);

// observers
key_compare key_comp() const;
value_compare value_comp() const;

// set operations
iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
template<class K> iterator find(const K& x);
template<class K> const_iterator find(const K& x) const;
size_type count(const key_type& x) const;
template<class K> size_type count(const K& x) const;

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);

pair<const_iterator, const_iterator> equal_range(const K& x) const;

};

template<class InputIterator,
class Compare = less<typename iterator_traits<InputIterator>::value_type>,
class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
multiset(InputIterator, InputIterator,
Compare = Compare(), Allocator = Allocator())
-> multiset<typename iterator_traits<InputIterator>::value_type, 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<typename iterator_traits<InputIterator>::value_type,
less<typename iterator_traits<InputIterator>::value_type>, Allocator>;

template<class Key, class Allocator>
multiset(initializer_list<Key>, Allocator) -> multiset<Key, less<Key>, Allocator>;

// 26.4.7.3, specialized algorithms
template<class Key, class Compare, class Allocator>
void swap(multiset<Key, Compare, Allocator>& x,
multiset<Key, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));

26.4.7.2 multiset constructors

explicit multiset(const Compare& comp, const Allocator& = Allocator());
1 Effects: Constructs an empty multiset using the specified comparison object and allocator.
2 Complexity: Constant.

template<class InputIterator>
multiset(InputIterator first, InputIterator last,
const Compare& comp = Compare(), const Allocator& = Allocator());
3 Effects: Constructs an empty multiset using the specified comparison object and allocator, and inserts elements from the range [first, last).
4 Complexity: Linear in \( N \) if the range [first, last) is already sorted using \( \text{comp} \) and otherwise \( N \log N \), where \( N \) is last - first.

26.4.7.3 multiset specialized algorithms

void swap(multiset<Key, Compare, Allocator>& x,
multiset<Key, Compare, Allocator>& y)
noexcept(noexcept(x.swap(y)));
1 Effects: As if by \( x.\text{swap}(y) \).

26.5 Unordered associative containers

26.5.1 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.
2 The exposition-only alias templates iter_key_t, iter_val_t, and iter_to_alloc_t defined in 26.4.1 may appear in deduction guides for unordered containers.
26.5.2 Header `<unordered_map>` synopsis

```cpp
#include <initializer_list>

namespace std {

// 26.5.4, 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;

// 26.5.5, 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);

namespace pmr {

using unordered_map = std::unordered_map<Key, T, Hash, Pred,
    polymorphic_allocator<pair<const Key, T>>;}

namespace pmr {

using unordered_multimap = std::unordered_multimap<Key, T, Hash, Pred,
    polymorphic_allocator<pair<const Key, T>>;}
}
}
26.5.3 Header <unordered_set> synopsis

```cpp
#include <initializer_list>

namespace std {
    // 26.5.6, class template unordered_set
    template<class Key,
        class Hash = hash<Key>,
        class Pred = equal_to<Key>,
        class Alloc = allocator<Key>>
    class unordered_set;

    // 26.5.7, 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);

    template<class Key, class Hash, class Pred, class Alloc>
    bool operator!=(const unordered_multiset<Key, Hash, Pred, Alloc>& a,
                    const unordered_multiset<Key, Hash, Pred, Alloc>& b);

    template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Key, Hash, Pred, Alloc>& x,
              unordered_set<Key, Hash, Pred, Alloc>& y) noexcept(noexcept(x.swap(y)));

    template<class Key, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Key, Hash, Pred, Alloc>& x,
              unordered_multiset<Key, Hash, Pred, Alloc>& y) noexcept(noexcept(x.swap(y)));

    namespace pmr {
        template<class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
        using unordered_set = std::unordered_set<Key, Hash, Pred,
                                               polymorphic_allocator<Key>>;

        template<class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
        using unordered_multiset = std::unordered_multiset<Key, Hash, Pred,
                                                          polymorphic_allocator<Key>>;
    }
}

26.5.4 Class template unordered_map

26.5.4.1 Class template unordered_map overview

An unordered_map is an unordered associative container that supports unique keys (an unordered_map contains at most one of each key value) and that associates values of another type mapped_type with the keys. The unordered_map class supports forward iterators.
An unordered_map satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 86). 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.

```cpp
namespace std {

    template<class Key, 
             class T, 
             class Hash = hash<Key>, 
             class Pred = equal_to<Key>, 
             class Allocator = allocator<pair<const Key, T>>>
    class unordered_map {
        public:
            // types
            using key_type = Key;
            using mapped_type = T;
            using value_type = pair<const Key, T>;
            using hasher = Hash;
            using key_equal = Pred;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 26.2
            using difference_type = implementation-defined; // see 26.2
            using iterator = implementation-defined; // see 26.2
            using const_iterator = implementation-defined; // see 26.2
            using local_iterator = implementation-defined; // see 26.2
            using const_local_iterator = implementation-defined; // see 26.2
            using node_type = unspecified;
            using insert_return_type = INSERT_RETURN_TYPE<iterator, node_type>;

            // 26.5.4.1

            unordered_map();
            explicit unordered_map(size_type n, 
                                    const hasher& hf = hasher(),
                                    const key_equal& eql = key_equal(),
                                    const allocator_type& a = allocator_type());

            template<class InputIterator>
            unordered_map(InputIterator f, InputIterator l, 
                            size_type n = see below,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());

            unordered_map(const unordered_map&);
            unordered_map(unordered_map&&);
            explicit unordered_map(const Allocator&);
            unordered_map(const unordered_map&, const Allocator&);
            unordered_map(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) { }

template<class InputIterator>
unordered_map(InputIterator f, InputIterator l, size_type n, const hasher& hf, const allocator_type& a) :
unordered_map(f, l, n, hf, key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const allocator_type& a) :
unordered_map(il, n, hasher(), key_equal(), a) { }
unordered_map(initializer_list<value_type> il, size_type n, const hasher& hf, const allocator_type& a) :
unordered_map(il, n, hf, key_equal(), a) { }
~unordered_map();
unordered_map& operator=(const unordered_map&); unordered_map& operator=(unordered_map&&) noexcept(allocator_traits<Allocator>::is_always_equal::value &&
is_nothrow_move_assignable_v<Hash> &&
is_nothrow_move_assignable_v<Pred>);
unordered_map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 26.5.4.4, modifiers
template<class... Args> pair<iterator, bool> emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
pair<iterator, bool> insert(const value_type& obj);
pair<iterator, bool> insert(value_type&& obj);
template<class P> pair<iterator, bool> insert(P&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class P> iterator insert(const_iterator hint, P&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
node_type extract(const_iterator position);
node_type extract(const key_type& x);
insert_return_type insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

template<class... Args>
pair<iterator, bool> try_emplace(const key_type& k, Args&&... args);
template<class... Args>
pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, const key_type& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);
template<class M>
pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);
template<class M>
pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
template<class M>
    iterator insert_or_assign(const_iterator hint, const key_type& k, M& obj);

template<class M>
    iterator insert_or_assign(const_iterator hint, key_type&& k, M& obj);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const_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;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// 26.5.4.3, element access
mapped_type& operator[](const key_type& k);
mapped_type& operator[](key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
local_iterator begin(size_type n) const;
local_iterator end(size_type n) const;
local_iterator cbegin(size_type n) const;
local_iterator cbegin(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_t<InputIterator>>>,
        class Pred = equal_to<iter_key_t<InputIterator>>>,
        class Allocator = allocator<iter_to_alloc_t<InputIterator>>>
unordered_map(InputIterator, InputIterator, typename see below::size_type = see below,
        Hash = Hash(), Pred = Pred(), Allocator = Allocator())
-> unordered_map<iter_key_t<InputIterator>, iter_val_t<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<const 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_t<InputIterator>, iter_val_t<InputIterator>, hash<iter_key_t<InputIterator>>,
        equal_to<iter_key_t<InputIterator>>, Allocator>;

template<class Key, class T, class Hash, class Allocator>
unordered_map(initializer_list<pair<const Key, T>>, typename see below::size_type, Hash,
        Allocator)
-> unordered_map<Key, T, Hash, equal_to<Key>, Allocator>;

// 26.5.4.5, swap
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.

26.5.4.2 unordered_map constructors [unord.map.cnstr]

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());

2 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.

26.5.4.3 unordered_map 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);
const mapped_type& at(const key_type& k) const;

3 Returns: A reference to \( x.second \), where \( x \) is the (unique) element whose key is equivalent to \( k \).

4 Throws: An exception object of type out_of_range if no such element is present.

26.5.4.4 unordered_map modifiers

template<class P>
pair<iterator, bool> insert(P&& obj);

1 Effects: Equivalent to: return emplace(std::forward<P>(obj));

Remarks: This signature shall not participate in overload resolution unless is_constructible_v<value_type, \&\&P> is true.

template<class P>
iterator insert(const_iterator hint, P&& obj);

3 Effects: Equivalent to: return emplace_hint(hint, std::forward<P>(obj));

Remarks: This signature shall not participate in overload resolution unless is_constructible_v<value_type, \&\&P> is true.

template<class... Args>
pair<iterator, bool> try_emplace(const key_type& k, Args\&\&... args);

4 Requires: value_type shall be EmplaceConstructible into unordered_map from piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)\...).

§ 26.5.4.4
Effects: If the map already contains an element whose key is equivalent to k, there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(k), forward_as_tuple(std::forward<Args>(args)...)..

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

template<class... Args>
pair<iterator, bool> try_emplace(key_type&& k, Args&&... args);
template<class... Args>
iterator try_emplace(const_iterator hint, key_type&& k, Args&&... args);

Requires: value_type shall be EmplaceConstructible into unordered_map from piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...)..

Effects: If the map already contains an element whose key is equivalent to k, there is no effect. Otherwise inserts an object of type value_type constructed with piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...)..

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

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);

Requires: is_assignable_v<mapped_type&, M&&> shall be true. value_type shall be EmplaceConstructible into unordered_map from k, std::forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with k, std::forward<M>(obj).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

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);

Requires: is_assignable_v<mapped_type&, M&&> shall be true. value_type shall be EmplaceConstructible into unordered_map from std::move(k), std::forward<M>(obj).

Effects: If the map already contains an element e whose key is equivalent to k, assigns std::forward<M>(obj) to e.second. Otherwise inserts an object of type value_type constructed with std::move(k), std::forward<M>(obj).

Returns: In the first overload, the bool component of the returned pair is true if and only if the insertion took place. The returned iterator points to the map element whose key is equivalent to k.

Complexity: The same as emplace and emplace_hint, respectively.

26.5.4.5 unordered_map swap

template<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)));

Effects: As if by x.swap(y).
26.5.5 Class template unordered_multimap

26.5.5.1 Class template unordered_multimap overview

An unordered_multimap is an unordered associative container that supports equivalent keys (an instance of unordered_multimap may contain multiple copies of each key value) and that associates values of another type mapped_type with the keys. The unordered_multimap class supports forward iterators.

An unordered_multimap satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 86). 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>.

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.

```cpp
namespace std {
    template<class Key,
             class T,
             class Hash = hash<Key>,
             class Pred = equal_to<Key>,
             class Allocator = allocator<pair<const Key, T>>>
    class unordered_multimap {
        public:
            // types
            using key_type = Key;
            using mapped_type = T;
            using value_type = pair<const Key, T>;
            using hasher = Hash;
            using key_equal = Pred;
            using allocator_type = Allocator;
            using pointer = typename allocator_traits<Allocator>::pointer;
            using const_pointer = typename allocator_traits<Allocator>::const_pointer;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = implementation-defined; // see 26.2
            using difference_type = implementation-defined; // see 26.2
            using iterator = implementation-defined; // see 26.2
            using const_iterator = implementation-defined; // see 26.2
            using local_iterator = implementation-defined; // see 26.2
            using const_local_iterator = implementation-defined; // see 26.2
            using node_type = unspecified;

            // 26.5.5.2, construct/copy/destroy
            unordered_multimap();
            explicit unordered_multimap(size_type n,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());

            template<class InputIterator>
            unordered_multimap(InputIterator f, InputIterator l,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());

            unordered_multimap(const unordered_multimap&);
            unordered_multimap(unordered_multimap&&);
            explicit unordered_multimap(const Allocator&);
            unordered_multimap(const unordered_multimap&, const Allocator&);
            unordered_multimap(unordered_multimap&, const Allocator&);
            unordered_multimap(initializer_list<value_type> il,
                size_type n = see below,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal()),
```
const allocator_type& a = allocator_type();
unordered_multimap(size_type n, const allocator_type& a) :
unordered_multimap(n, hasher(), key_equal(), a) { }
unordered_multimap(size_type n, const hasher& hf, const allocator_type& a) :
unordered_multimap(n, hf, key_equal(), a) { }
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n, const allocator_type& a) :
unordered_multimap(f, l, n, hasher(), key_equal(), a) { }
template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l, size_type n, const hasher& hf,
const allocator_type& a) :
unordered_multimap(f, l, n, hf, key_equal(), a) { }
unordered_multimap(initializer_list<value_type> il, size_type n, const allocator_type& a) :
unordered_multimap(il, n, hasher(), key_equal(), a) { }
unordered_multimap(initializer_list<value_type> il, size_type n, const hasher& hf,
const allocator_type& a) :
unordered_multimap(il, n, hf, key_equal(), a) { }
-unordered_multimap();
unordered_multimap operator=(const unordered_multimap&);
unordered_multimap operator=(unordered_multimap&&)
    noexcept(algorithm_traits<Allocator>::is_always_equal::value &&
    is_nothrow_move_assignable_v<Hash> &&
    is_nothrow_move_assignable_v<Pred>);
unordered_multimap operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// 26.5.5.3, modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
template<class P> iterator insert(P&& obj);
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);
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(algorithm_traits<Allocator>::is_always_equal::value &&
    is_nothrow_swappable_v<Hash> &&
    is_nothrow_swappable_v<Pred>);
void clear() noexcept;

template<class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>& source);

template<class H2, class P2>
void merge(unordered_multimap<Key, T, H2, P2, Allocator>&& source);

template<class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>& source);

template<class H2, class P2>
void merge(unordered_map<Key, T, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// map operations
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
};

template<class InputIterator,
    class Hash = hash<iter_key_t<InputIterator>>,
    class Pred = equal_to<iter_key_t<InputIterator>>,
    class Allocator = allocator<iter_to_alloc_t<InputIterator>>>
unordered_multimap(InputIterator, InputIterator,
typename see below::size_type = see below,
Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_multimap<iter_key_t<InputIterator>, iter_val_t<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<const 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_t<InputIterator>, iter_val_t<InputIterator>,
    hash<iter_key_t<InputIterator>>,
    equal_to<iter_key_t<InputIterator>>, Allocator>;}
template<class InputIterator, class Allocator>
unordered_multimap(InputIterator, InputIterator, Allocator)
-> unordered_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>,
hash<iter_key_t<InputIterator>>,
equal_to<iter_key_t<InputIterator>>, Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_multimap(InputIterator, InputIterator, typename
see below::size_type, Hash,
Allocator)
-> unordered_multimap<iter_key_t<InputIterator>, iter_val_t<InputIterator>, Hash,
equal_to<iter_key_t<InputIterator>>, Allocator>;

template<class Key, class T, class Allocator>
unordered_multimap(initializer_list<pair<const Key, T>>, typename
see below::size_type,
Allocator)
-> unordered_multimap<Key, T, hash<Key>, equal_to<Key>, Allocator>;

// 26.5.5.4, swap
template<class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
unordered_multimap<Key, T, Hash, Pred, Alloc>& y)
noexcept(noexcept(x.swap(y)));

A size_type parameter type in an unordered_multimap deduction guide refers to the size_type member type of the type deduced by the deduction guide.

26.5.5.2 unordered_multimap 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());

Effects: Constructs an empty unordered_multimap using the specified hash function, key equality predicate, and allocator, and using at least n buckets. For the default constructor, the number of buckets is implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>
unordered_multimap(InputIterator f, InputIterator l,
size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

unordered_multimap(initializer_list<value_type> il,
size_type n = see below,
const hasher& hf = hasher(),
const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multimap using the specified hash function, key equality predicate, and allocator, and using at least n buckets. If n is not provided, the number of buckets is implementation-defined. Then inserts elements from the range [f, l) for the first form, or from the range [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.
Complexity: Average case linear, worst case quadratic.

26.5.5.3 unordered_multimap modifiers

```cpp
template<class P>
iterator insert(P&& obj);
```

**Effects:** Equivalent to: `return emplace(std::forward<P>(obj));`;

**Remarks:** This signature shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.

```cpp
template<class P>
iterator insert(const_iterator hint, P&& obj);
```

**Effects:** Equivalent to: `return emplace_hint(hint, std::forward<P>(obj));`;

**Remarks:** This signature shall not participate in overload resolution unless `is_constructible_v<value_type, P&&>` is `true`.

26.5.5.4 unordered_multimap swap

```cpp
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)));
```

**Effects:** As if by `x.swap(y)`.

26.5.6 Class template unordered_set

26.5.6.1 Class template unordered_set 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 allocatort-aware container (Table 86). 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.

```cpp
namespace std {
  template<class Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<Key>>
  class unordered_set {
    public:
      // types
      using key_type = Key;
      using value_type = Key;
      using hasher = Hash;
      using key_equal = Pred;
      using allocator_type = Allocator;
      using pointer = typename allocator_traits<Allocator>::pointer;
      using const_pointer = typename allocator_traits<Allocator>::const_pointer;
      using reference = value_type&;
      using const_reference = const value_type&;
      using size_type = implementation_defined; // see 26.2
      using difference_type = implementation_defined; // see 26.2
      using iterator = implementation_defined; // see 26.2
      using const_iterator = implementation_defined; // see 26.2
      using local_iterator = implementation_defined; // see 26.2
      using const_local_iterator = implementation_defined; // see 26.2
  }
}
```
using node_type = unspecified;
using insert_return_type = INSERT_RETURN_TYPE<iterator, node_type>;

// 26.5.6.2, 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(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, hasher(), key_equal(), a) {}
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&);

template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_set<Key, H2, P2, Allocator>&& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>& source);
template<class H2, class P2>
void merge(unordered_multiset<Key, H2, P2, Allocator>&& source);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// set operations
iterator find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
}
template<class InputIterator, 
    class Hash = hash<typename iterator_traits<InputIterator>::value_type>, 
    class Pred = equal_to<typename iterator_traits<InputIterator>::value_type>, 
    class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
unordered_set(InputIterator, InputIterator, typename see below::size_type = see below, 
    Hash = Hash(), Pred = Pred(), Allocator = Allocator())
    -> unordered_set<typename iterator_traits<InputIterator>::value_type, 
    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<typename iterator_traits<InputIterator>::value_type, 
    hash<typename iterator_traits<InputIterator>::value_type>, 
    equal_to<typename iterator_traits<InputIterator>::value_type>, 
    Allocator>;

template<class InputIterator, class Hash, class Allocator>
unordered_set(InputIterator, InputIterator, typename see below::size_type, Hash, 
    equal_to<typename iterator_traits<InputIterator>::value_type>, 
    Allocator) 
    -> unordered_set<typename iterator_traits<InputIterator>::value_type, Hash, 
    equal_to<typename iterator_traits<InputIterator>::value_type>, 
    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>;

// 26.5.6.3, 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);

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.

26.5.6.2 unordered_set constructors

unordered_set() : unordered_set(size_type(see below)) { }  
explicit unordered_set(size_type n, 
    const hasher& hf = hasher(), 
    const key_equal& eql = key_equal(), 
    const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_set using the specified hash function, key equality predicate, 
and allocator, and using at least n buckets. For the default constructor, the number of buckets is 
implementation-defined. max_load_factor() returns 1.0.

Complexity: Constant.

template<class InputIterator>
unordered_set(InputIterator f, InputIterator l, 
    size_type n = see below, 
    const hasher& hf = hasher(), 
    const key_equal& eql = key_equal(),
const allocator_type& a = allocator_type();
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());
3
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 [il.begin(), il.end()) for the second form. max_load_factor() returns 1.0.

Complexity: Average case linear, worst case quadratic.

26.5.6.3 unordered_set 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)));
1
Effects: As if by x.swap(y).

26.5.7 Class template unordered_multiset

26.5.7.1 Class template unordered_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 86). 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 {
    public:
        // types
        using key_type = Key;
        using value_type = Key;
        using hasher = Hash;
        using key_equal = Pred;
        using allocator_type = Allocator;
        using pointer = typename allocator_traits<Allocator>::pointer;
        using const_pointer = typename allocator_traits<Allocator>::const_pointer;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = implementation-defined; // see 26.2
        using difference_type = implementation-defined; // see 26.2
        using iterator = implementation-defined; // see 26.2
        using const_iterator = implementation-defined; // see 26.2
        using local_iterator = implementation-defined; // see 26.2
        using const_local_iterator = implementation-defined; // see 26.2
        using node_type = unspecified;

§ 26.5.7.1 843
// 26.5.7.2, 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 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(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();
unordered_multiset& operator=(const unordered_multiset&);
unordered_multiset& operator=(unordered_multiset&&)
  noexcept(allocator_traits<Allocator>::is_always_equal::value &&
    is_nothrow_move_assignable_v<Hash> &&
    is_nothrow_move_assignable_v<Pred>);
unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const noexcept;

// iterators
iterator begin() noexcept;
const_iterator begin() const noexcept;
iterator end() noexcept;
const_iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;

// capacity
[[nodiscard]] bool empty() const noexcept;
size_type size() const noexcept;
size_type max_size() const noexcept;

// modifiers
template<class... Args> iterator emplace(Args&&... args);
template<class... Args> iterator emplace_hint(const_iterator position, Args&&... args);
iterator insert(const value_type& obj);
iterator insert(value_type&& obj);
iterator insert(const_iterator hint, const value_type& obj);
iterator insert(const_iterator hint, value_type&& obj);
template<class InputIterator> void insert(InputIterator first, InputIterator last);
void insert(initializer_list<value_type>);
node_type extract(const_iterator position);
node_type extract(const key_type& x);
iterator insert(node_type&& nh);
iterator insert(const_iterator hint, node_type&& nh);

iterator erase(iterator position);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void swap(unordered_multiset&);

// 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;
size_type count(const key_type& k) const;
pair<iterator, iterator> equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const noexcept;
size_type max_bucket_count() const noexcept;
size_type bucket_size(size_type n) const;
local_iterator begin(size_type n) const;
local_iterator begin(const key_type& k) const;
local_iterator end(size_type n) const;
local_iterator end(const key_type& k) const;
local_iterator cbegin(size_type n) const;
local_iterator cbegin(const key_type& k) const;
local_iterator cend(size_type n) const;
local_iterator cend(const key_type& k) const;

// hash policy
float load_factor() const noexcept;
float max_load_factor() const noexcept;
void max_load_factor(float z);
void rehash(size_type n);
void reserve(size_type n);
template<class InputIterator, 
   class Hash = hash<typename iterator_traits<InputIterator>::value_type>, 
   class Pred = equal_to<typename iterator_traits<InputIterator>::value_type>, 
   class Allocator = allocator<typename iterator_traits<InputIterator>::value_type>>
unordered_multiset(InputIterator, InputIterator, 
   see below::size_type = see below, 
   Hash = Hash(), Pred = Pred(), Allocator = Allocator())
   -> unordered_multiset<typename iterator_traits<InputIterator>::value_type, 
   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<typename iterator_traits<InputIterator>::value_type, 
   hash<typename iterator_traits<InputIterator>::value_type>, 
   equal_to<typename iterator_traits<InputIterator>::value_type>, 
   Allocator>;

template<class T, class Hash, class Allocator>
unordered_multiset(initializer_list<T>, typename 
   see below::size_type, Hash, 
   equal_to<typename iterator_traits<InputIterator>::value_type>, 
   Allocator>;

// 26.5.7.3, 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.

26.5.7.2 unordered_multiset constructors

unordered_multiset() : unordered_multiset(size_type(see below)) {} explicit unordered_multiset(size_type n, 
   const hasher& hf = hasher(),
   const key_equal& eql = key_equal(),
   const allocator_type& a = allocator_type());

Effects: Constructs an empty unordered_multiset using the specified hash function, key equality predicate, and allocator, and using at least n buckets. For the default constructor, the number of buckets is implementation-defined. 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 \( [i, 1) \) 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.

26.5.7.3 unordered_multiset 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)));

Effects: As if by \( x.swap(y) \).

26.6 Container adaptors

26.6.1 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:

(4.1) — It has an InputIterator template parameter and a type that does not qualify as an input iterator is deduced for that parameter.

(4.2) — It has a Compare template parameter and a type that qualifies as an allocator is deduced for that parameter.

(4.3) — It has a Container template parameter and a type that qualifies as an allocator is deduced for that parameter.

(4.4) — It has an Allocator template parameter and a type that does not qualify as an allocator is deduced for that parameter.

(4.5) — It has both Container and Allocator template parameters, and uses_allocator_v<Container, Allocator> is false.

26.6.2 Header <queue> synopsis

#include <initializer_list>

namespace std {

template<class T, class Container = deque<T>> class queue;
template<class T, class Container = vector<T>,
class Compare = less<typename Container::value_type>>
class priority_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);

§ 26.6.2
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 Compare>
void swap(priority_queue<T, Container, Compare>& x, priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

26.6.3 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)));
}

26.6.4 Class template queue

26.6.4.1 queue definition

Any sequence container supporting operations \texttt{front()}, \texttt{back()}, \texttt{push\_back()} and \texttt{pop\_front()} can be used to instantiate queue. In particular, \texttt{list} (26.3.10) and \texttt{deque} (26.3.8) can be used.

namespace std {
  template<class T, class Container = deque<T>>
  class queue {
    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:
      explicit queue(const Container&);
      explicit queue(Container& = Container());
      template<class Alloc> explicit queue(const Alloc&);
      template<class Alloc> queue(const Container&, const Alloc&);
  }
template<class Alloc> queue(const queue&, const Alloc&);
template<class Alloc> queue(queue&&, const Alloc&);

[[nodiscard]] bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
reference front() { return c.front(); }
const_reference front() const { return c.front(); }
reference back() { return c.back(); }
const_reference back() const { return c.back(); }
void push(const value_type& x) { c.push_back(x); }
void push(value_type&& x) { c.push_back(std::move(x)); }
void push(const value_type&& x) { c.push_back(std::move(x)); }
template<class... Args>
    decltype(auto) emplace(Args&&... args)
    { return c.emplace_back(std::forward<Args>(args)...); }
void pop() { c.pop_front(); }
void swap(queue& q) noexcept(is_nothrow_swappable_v<Container>)
    { using std::swap; swap(c, q.c); }
};
template<class Container>
    queue(Container) -> queue<typename Container::value_type, Container>;
template<class Container, class Allocator>
    queue(Container, Allocator) -> queue<typename Container::value_type, Container>;
template<class T, class Container>
    void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));

§ 26.6.4.2  queue constructors

explicit queue(const Container& cont);
  Effects: Initializes c with cont.

explicit queue(Container&& cont = Container());
  Effects: Initializes c with std::move(cont).

26.6.4.3 queue constructors with allocators

1 If uses_allocator_v<container_type, Alloc> is false the constructors in this subclause shall not participate in overload resolution.

template<class Alloc> explicit queue(const Alloc& a);
  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.c 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.c) as the first argument and a as the second argument.
26.6.4.4 queue operators

```cpp
template<class T, class Container>
bool operator==(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c == y.c.
```

```cpp
template<class T, class Container>
bool operator!=(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c != y.c.
```

```cpp
template<class T, class Container>
bool operator<(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c < y.c.
```

```cpp
template<class T, class Container>
bool operator<=(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c <= y.c.
```

```cpp
template<class T, class Container>
bool operator>(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c > y.c.
```

```cpp
template<class T, class Container>
bool operator>=(const queue<T, Container>& x, const queue<T, Container>& y);

Returns: x.c >= y.c.
```

26.6.4.5 queue specialized algorithms

```cpp
template<class T, class Container>
void swap(queue<T, Container>& x, queue<T, Container>& y) noexcept(noexcept(x.swap(y)));

Remarks: This function shall not participate in overload resolution unless is_swappable_v<Container>
is true.
```

Effects: As if by x.swap(y).

26.6.5 Class template priority_queue

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 (26.3.11) and deque (26.3.8)
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 (28.7).

```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(const Compare& x, const Container&);
            explicit priority_queue(const Compare& x = Compare(), Container&& = Container());
```
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());

template<class Alloc> explicit priority_queue(const Alloc&);

template<class Alloc> priority_queue(const Compare&, const Alloc&);

template<class Alloc> priority_queue(const Compare&, const Container&, const Alloc&);

template<class Alloc> priority_queue(const priority_queue&, const Alloc&);

template<class Alloc> priority_queue(priority_queue&&, const Alloc&);

[[nodiscard]] bool empty() const { return c.empty(); }
size_type size() const { return c.size(); }
const_reference top() const { return c.front(); }
void push(const value_type& x);
void push(value_type&& x);

template<class... Args> void emplace(Args&&... args);

void pop();
void swap(priority_queue& q) noexcept(is_nothrow_swappable_v<Container> &&
    is_nothrow_swappable_v<Compare>)
    { using std::swap; swap(c, q.c); swap(comp, q.comp); }
};

template<class Compare, class Container>
    priority_queue(Compare, Container)
    -> priority_queue<typename Container::value_type, Container, Compare>;

template<class InputIterator,
          class Compare = less<typename iterator_traits<InputIterator>::value_type>,
          class Container = vector<typename iterator_traits<InputIterator>::value_type>>
    priority_queue(InputIterator, InputIterator, Compare = Compare(), Container = Container())
    -> priority_queue<typename iterator_traits<InputIterator>::value_type, Container, Compare>;

template<class Compare, class Container, class Allocator>
    priority_queue(Compare, Container, Allocator)
    -> priority_queue<typename Container::value_type, Container, Compare>;

// no equality is provided

template<class T, class Container, class Compare>
    void swap(priority_queue<T, Container, Compare>& x,
        priority_queue<T, Container, Compare>& y) noexcept(noexcept(x.swap(y)));

template<class T, class Container, class Compare, class Alloc>
struct uses_allocator<priority_queue<T, Container, Compare>, Alloc>
    : uses_allocator<Container, Alloc>::type { }
};

26.6.5.1 priority_queue constructors [priqueue.cons]

priority_queue(const Compare& x, const Container& y);
explicit priority_queue(const Compare& x = Compare(), Container&& y = Container());

1 Requires: x shall define a strict weak ordering (28.7).
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());

Requires: x shall define a strict weak ordering (28.7).
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).

26.6.5.2 priority_queue constructors with allocators

If uses_allocator_v<container_type, Alloc> is false the constructors in this subclause shall not participate
in overload resolution.

template<class Alloc> explicit priority_queue(const Alloc& a);
Effects: Initializes c with a and value-initializes comp.

template<class Alloc> priority_queue(const Compare& compare, const Alloc& a);
Effects: Initializes c with a and initializes comp with compare.

template<class Alloc>
priority_queue(const Compare& compare, const Container& cont, const Alloc& a);
Effects: Initializes c with cont as the first argument and a as the second argument, and initializes
comp with compare; calls make_heap(c.begin(), c.end(), comp).

template<class Alloc>
priority_queue(const Compare& compare, Container&& cont, const Alloc& a);
Effects: Initializes c with std::move(cont) as the first argument and a as the second argument, and initializes
comp with compare; calls make_heap(c.begin(), c.end(), comp).

template<class Alloc> priority_queue(const priority_queue& q, const Alloc& a);
Effects: Initializes c with q.c as the first argument and a as the second argument, and initializes
comp with q.comp.

template<class Alloc> priority_queue(priority_queue&& q, const Alloc& a);
Effects: Initializes c with std::move(q.c) as the first argument and a as the second argument, and initializes
comp with std::move(q.comp).

26.6.5.3 priority_queue 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();
26.6.5.4 priority_queue 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)));
```

Remarks: This function shall not participate in overload resolution unless is_swappable_v<Container> is true and is_swappable_v<Compare> is true.

Effects: As if by x.swap(y).

26.6.6 Class template stack

Any sequence container supporting operations back(), push_back() and pop_back() can be used to instantiate stack. In particular, vector (26.3.11), list (26.3.10) and deque (26.3.8) can be used.

26.6.6.1 stack 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:
        explicit stack(const Container&);
        explicit stack(Container&& = 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 { };}
```
26.6.6.2  stack constructors

explicit stack(const Container& cont);
1  
  Effects: Initializes c with cont.
explicit stack(Container&& cont = Container());
2  
  Effects: Initializes c with std::move(cont).

26.6.6.3  stack 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);
1  
  Effects: Initializes c with a.
template<class Alloc> stack(const container_type& cont, const Alloc& a);
3  
  Effects: Initializes c with cont as the first argument and a as the second argument.
template<class Alloc> stack(container_type&& cont, const Alloc& a);
4  
  Effects: Initializes c with std::move(cont) as the first argument and a as the second argument.
template<class Alloc> stack(const stack& s, const Alloc& a);
5  
  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);
6  
  Effects: Initializes c with std::move(s.c) as the first argument and a as the second argument.

26.6.6.4  stack operators

template<class T, class Container>
  bool operator==(const stack<T, Container>& x, const stack<T, Container>& y);
1  
  Returns: x.c == y.c.
template<class T, class Container>
  bool operator!=(const stack<T, Container>& x, const stack<T, Container>& y);
2  
  Returns: x.c != y.c.
template<class T, class Container>
  bool operator<(const stack<T, Container>& x, const stack<T, Container>& y);
3  
  Returns: x.c < y.c.
template<class T, class Container>
  bool operator<=(const stack<T, Container>& x, const stack<T, Container>& y);
4  
  Returns: x.c <= y.c.
template<class T, class Container>
  bool operator>(const stack<T, Container>& x, const stack<T, Container>& y);
5  
  Returns: x.c > y.c.
template<class T, class Container>
  bool operator>=(const stack<T, Container>& x, const stack<T, Container>& y);
6  
  Returns: x.c >= y.c.

26.6.6.5  stack specialized algorithms

template<class T, class Container>
  void swap(stack<T, Container>& x, stack<T, Container>& y) noexcept(noexcept(x.swap(y))); 1  
  Remarks: This function shall not participate in overload resolution unless is_swappable_v<Container> is true.
2  
  Effects: As if by x.swap(y).
27  Iterators library

27.1  General

This Clause describes components that C++ programs may use to perform iterations over containers (Clause 26), streams (30.7), and stream buffers (30.6).

The following subclauses describe iterator requirements, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 92.

Table 92 — Iterators library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>27.2</td>
<td>Requirements</td>
</tr>
<tr>
<td>27.4</td>
<td>Iterator primitives &lt;iterator&gt;</td>
</tr>
<tr>
<td>27.5</td>
<td>Predefined iterators</td>
</tr>
<tr>
<td>27.6</td>
<td>Stream iterators</td>
</tr>
</tbody>
</table>

27.2  Iterator requirements

27.2.1 In general

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) 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 \( T \). An iterator \( i \) for which the expression \( (\ast i).m \) is well-defined supports the expression \( i\rightarrow m \) with the same semantics as \( (\ast i).m \). For every iterator type \( X \) for which equality is defined, there is a corresponding signed integer type called the difference type of the iterator.

Since iterators are an abstraction of pointers, their semantics is 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 five categories of iterators, according to the operations defined on them: input iterators, output iterators, forward iterators, bidirectional iterators and random access iterators, as shown in Table 93.

Table 93 — Relations among iterator categories

| Random Access → Bidirectional → Forward → Input → Output |

3 Forward iterators satisfy all the requirements of input iterators and can be used wherever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of forward iterators and can be used wherever 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.

4 Iterators that further satisfy the requirements of output iterators are called mutable iterators. Nonmutable iterators are referred to as constant iterators.

5 In addition to the requirements in this subclause, the nested typedef-names specified in 27.4.1 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]

6 Iterators that further satisfy the requirement that, for integral values \( n \) and dereferenceable iterator values \( a \) and \( (a + n) \), \( \ast(a + n) \) is equivalent to \( \ast(\text{addressof}(\ast a) + n) \), are called contiguous iterators. [Note: For example, the type “pointer to int” is a contiguous iterator, but reverse_iterator<int *> is not.
For a valid iterator range \([a, b]\) with dereferenceable \(a\), the corresponding range denoted by pointers is 
\[
\text{[addressof(*a), addressof(*a) + (b - a)]}; \ b \text{ might not be dereferenceable. — end note}
\]

Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of
the array, so for any iterator type there is an iterator value that points past the last element of a corresponding
sequence. These values are called past-the-end values. Values of an iterator \(i\) for which the expression \(*i\) is
defined are called dereferenceable. The library never assumes that past-the-end values are dereferenceable.
Iterators can also have singular values that are not associated with any sequence. [Example: After the
declaration of an uninitialized pointer \(x\) (as with \texttt{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 DefaultConstructible
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.

An iterator \(j\) is called reachable from an iterator \(i\) if and only if there is a finite sequence of applications of
the expression ++\(i\) that makes \(i == j\). If \(j\) is reachable from \(i\), they refer to elements of the same sequence.

Most of the library’s algorithmic templates that operate on data structures have interfaces that use ranges.
A range is a pair of iterators that designate the beginning and end of the computation. A range \([i, j]\) is an
empty range; in general, a range \([i, j]\) refers to the elements in the data structure starting with the element
pointed to by \(i\) and up to but not including the element pointed to by \(j\). Range \([i, j]\) is valid if and only if
\(j\) is reachable from \(i\). The result of the application of functions in the library 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 for the iterators do not have a complexity column.

Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

An invalid iterator is an iterator that may be singular.\(^{264}\)

In the following sections, \(a\) and \(b\) denote values of type \(X\) or \texttt{const X} difference_type and reference refer
to the types \texttt{iterator_traits<X>::difference_type} and \texttt{iterator_traits<X>::reference}, respectively,
\(n\) denotes a value of \texttt{difference_type}, \(u, \ tmp, \) and \(m\) denote identifiers, \(r\) denotes a value of \texttt{X&}, \(t\) denotes a
value of value type \(T, o\) denotes a value of some type that is writable to the output iterator. [Note: For an
iterator type \(X\) there must be an instantiation of \texttt{iterator_traits<X>} (27.4.1). — end note]

### 27.2.2 Iterator

The Iterator requirements form the basis of the iterator concept taxonomy; every iterator satisfies the Iterator
requirements. This set of requirements specifies operations for dereferencing and incrementing an
iterator. Most algorithms will require additional operations to read (27.2.3) or write (27.2.4) values, or to
provide a richer set of iterator movements (27.2.5, 27.2.6, 27.2.7).

A type \(X\) satisfies the Iterator requirements if:

\(^{(2.1)}\) — \(X\) satisfies the CopyConstructible, CopyAssignable, and Destructible requirements (20.5.3.1)
and traits of type \(X\) are swappable (20.5.3.2), and

\(^{(2.2)}\) — the expressions in Table 94 are valid and have the indicated semantics.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*r)</td>
<td>unspecified</td>
<td></td>
<td>Requires: (r) is dereferenceable.</td>
</tr>
<tr>
<td>++(r)</td>
<td>(X&amp;)\</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^{264}\) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.
27.2.3 Input iterators

A class or pointer type \( X \) satisfies the requirements of an input iterator for the value type \( T \) if \( X \) satisfies the Iterator (27.2.2) and EqualityComparable (Table 20) requirements and the expressions in Table 95 are valid and have the indicated semantics.

In Table 95, 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 \( \text{find}(a, b, x) \) 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 \( p \)).

---

Table 95 — Input iterator requirements (in addition to Iterator)

<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>Requires: ( a, b ) is in the domain of ( == ).</td>
</tr>
<tr>
<td>( *a )</td>
<td>reference, convertible to ( T )</td>
<td></td>
<td>Requires: ( a ) is dereferenceable. The expression ( \text{(void)}*a ) is equivalent to ( *a ). If ( a == b ) and ( (a, b) ) is in the domain of ( == ) then ( *a ) is equivalent to ( *b ).</td>
</tr>
<tr>
<td>( a-&gt;m )</td>
<td>( (*a).m )</td>
<td></td>
<td>Requires: ( a ) is dereferenceable.</td>
</tr>
<tr>
<td>++( r )</td>
<td>( *r )</td>
<td></td>
<td>Requires: ( r ) is dereferenceable. Postconditions: ( r ) is dereferenceable or ( r ) is past-the-end; any copies of the previous value of ( r ) are no longer required either to be dereferenceable or to be in the domain of ( == ).</td>
</tr>
<tr>
<td>( \text{(void)}r++ )</td>
<td>equivalent to ( \text{(void)}++r )</td>
<td></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>

---

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 CopyAssignable type (Table 26). These algorithms can be used with istreams as the source of the input data through the \text{istream_iterator} class template.

27.2.4 Output iterators

A class or pointer type \( X \) satisfies the requirements of an output iterator if \( X \) satisfies the Iterator requirements (27.2.2) and the expressions in Table 96 are valid and have the indicated semantics.
Table 96 — Output iterator requirements (in addition to Iterator)

<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. Postconditions: r is incrementable.</td>
<td></td>
</tr>
<tr>
<td>++r</td>
<td>X&amp;</td>
<td>&amp;r == &amp;++r. Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
<td></td>
</tr>
<tr>
<td>r++</td>
<td>convertible to const X&amp;</td>
<td>Remarks: After this operation r is not required to be dereferenceable. Postconditions: r is incrementable.</td>
<td></td>
</tr>
<tr>
<td>*r++ = o</td>
<td>result is not used</td>
<td>Remarks: After this operation r is not required to be dereferenceable. Postconditions: r 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. Algorithms that take output iterators can be used with ostream as the destination for placing data through the ostream_iterator class as well as with insert iterators and insert pointers. — end note]

27.2.5 Forward iterators

A class or pointer type X satisfies the requirements of a forward iterator if

1. X satisfies the requirements of an input iterator (27.2.3),
2. X satisfies the DefaultConstructible requirements (20.5.3.1),
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 97 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 97 — Forward iterator requirements (in addition to input iterator)

<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</td>
<td>{ X tmp = r; const X&amp; ++r; return tmp; }</td>
<td></td>
</tr>
<tr>
<td>*r++</td>
<td>reference</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 If a and b are equal, then either a and b are both dereferenceable or else neither is dereferenceable.
6 If a and b are both dereferenceable, then a == b if and only if *a and *b are bound to the same object.

27.2.6 Bidirectional iterators

A class or pointer type X satisfies the requirements of a bidirectional iterator if, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 98.

Table 98 — Bidirectional iterator requirements (in addition to forward iterator)

<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></td>
<td>Requires: there exists s such that r == ++s. Postconditions: r is dereferenceable. --(++r) == r. --r == --s implies r == s. &amp;r == &amp;--r.</td>
</tr>
<tr>
<td>r--</td>
<td>convertible to</td>
<td>{ X tmp = r; const X&amp; --r; return tmp; }</td>
<td></td>
</tr>
<tr>
<td>*r--</td>
<td>reference</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2 [Note: Bidirectional iterators allow algorithms to move iterators backward as well as forward. — end note]

27.2.7 Random access iterators

A class or pointer type X satisfies the requirements of a random access iterator if, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 99.

Table 99 — Random access iterator requirements (in addition to bidirectional iterator)

<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; a + n == n + a. }</td>
<td></td>
</tr>
<tr>
<td>n + a</td>
<td></td>
<td>return tmp += n; }</td>
<td></td>
</tr>
</tbody>
</table>

§ 27.2.7
### Random access iterator requirements (in addition to bidirectional iterator) (continued)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r -= n</code></td>
<td>X&amp;</td>
<td>return r += -n;</td>
<td>Requires: the absolute value of n is in the range of representable values of difference_type.</td>
</tr>
<tr>
<td><code>a - n</code></td>
<td>X</td>
<td><code>{ X tmp = a; return tmp -= n; }</code></td>
<td></td>
</tr>
<tr>
<td><code>b - a</code></td>
<td>difference_type</td>
<td>return n</td>
<td>Requires: there exists a value n of type difference_type such that a + n == b. b == a + (b - a).</td>
</tr>
<tr>
<td><code>a[n]</code></td>
<td>convertible to reference</td>
<td>*(a + n)</td>
<td></td>
</tr>
<tr>
<td><code>a &lt; b</code></td>
<td>contextually convertible to bool</td>
<td>b - a &gt; 0</td>
<td>&lt; is a total ordering relation</td>
</tr>
<tr>
<td><code>a &gt; b</code></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><code>a &gt;= b</code></td>
<td>contextually convertible to bool</td>
<td>!(a &lt; b)</td>
<td></td>
</tr>
<tr>
<td><code>a &lt;= b</code></td>
<td>contextually convertible to bool</td>
<td>!(a &gt; b)</td>
<td></td>
</tr>
</tbody>
</table>

#### 27.3 Header `<iterator>` synopsis

```cpp
namespace std {

// 27.4, primitives
template<class Iterator> struct iterator_traits;
template<class T> struct iterator_traits<T*>;

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 { }; 

// 27.4.3, iterator operations
template<class InputIterator, class Distance>
constexpr void advance(InputIterator& i, Distance n);
template<class InputIterator>
constexpr typename iterator_traits<InputIterator>::difference_type distance(InputIterator first, InputIterator last);
template<class InputIterator>
constexpr InputIterator next(InputIterator x, 
    typename iterator_traits<InputIterator>::difference_type n = 1);
template<class BidirectionalIterator>
constexpr BidirectionalIterator prev(BidirectionalIterator x, 
    typename iterator_traits<BidirectionalIterator>::difference_type n = 1);

// 27.5, predefined iterators
template<class Iterator> class reverse_iterator;
```

§ 27.3
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 Container> class back_insert_iterator;

template<class Container>
back_insert_iterator<Container> back_inserter(Container& x);

template<class Container> class front_insert_iterator;

template<class Container>
front_insert_iterator<Container> front_inserter(Container& x);

template<class Container> class insert_iterator;

template<class Container>
insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

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 Iterator>
constexpr move_iterator<Iterator> operator+(typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

template<class Iterator>
constexpr move_iterator<Iterator> make_move_iterator(Iterator i);

// 27.6, stream iterators

template<class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;

template<class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
    const istream_iterator<T,charT,traits,Distance>& y);

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 charT, class traits = char_traits<charT>>
class istreambuf_iterator;

template<class charT, class traits>
bool operator==(const istreambuf_iterator<charT,traits>& a,
    const istreambuf_iterator<charT,traits>& b);

template<class charT, class traits>
bool operator!=(const istreambuf_iterator<charT,traits>& a,
    const istreambuf_iterator<charT,traits>& b);

template<class charT, class traits = char_traits<charT>>
class ostreambuf_iterator;

// 27.7, range access

template<class C> constexpr auto begin(C& c) -> decltype(c.begin());

template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());

template<class C> constexpr auto end(C& c) -> decltype(c.end());

template<class C> constexpr auto end(const C& c) -> decltype(c.end());

template<class T, size_t N> constexpr T* begin(T (&array)[N]) noexcept;

template<class T, size_t N> constexpr T* end(T (&array)[N]) noexcept;

template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c));

template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
    -> decltype(std::end(c));

template<class C> constexpr auto rbegin(C& c) -> decltype(c.rbegin());

template<class C> constexpr auto rbegin(const C& c) -> decltype(c.rbegin());

template<class C> constexpr auto rend(C& c) -> decltype(c.rend());

template<class C> constexpr auto rend(const C& c) -> decltype(c.rend());

template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);

template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);

template<class C> constexpr reverse_iterator<const C*> rbegin(initializer_list<C> il);

template<class C> constexpr reverse_iterator<const C*> rend(initializer_list<C> il);

template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));

template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));
27.4 Iterator primitives

To simplify the task of defining iterators, the library provides several classes and functions:

27.4.1 Iterator traits

To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if `Iterator` is the type of an iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
```

be defined as the iterator’s difference type, value type and iterator category, respectively. In addition, the types

```
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

shall be defined as the iterator’s reference and pointer types, that is, for an iterator object `a`, the same type as the type of `*a` and `a->`, respectively. In the case of an output iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

may be defined as void.

If `Iterator` has valid (17.9.2) member types `difference_type`, `value_type`, `pointer`, `reference`, and `iterator_category`, `iterator_traits<Iterator>` shall have the following as publicly accessible members:

```
using difference_type = typename Iterator::difference_type;
using value_type = typename Iterator::value_type;
using pointer = typename Iterator::pointer;
using reference = typename Iterator::reference;
using iterator_category = typename Iterator::iterator_category;
```

Otherwise, `iterator_traits<Iterator>` shall have no members by any of the above names.

It is specialized for pointers as

```
namespace std {
    template<class T> struct iterator_traits<T*> {
        using difference_type = ptrdiff_t;
        using value_type = remove_cv_t<T>;
        using pointer = T*;
        using reference = T&;
        using iterator_category = random_access_iterator_tag;
    };
}
```

[Example: To implement a generic reverse function, a C++ program can do the following:

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last) {
    typename iterator_traits<BidirectionalIterator>::difference_type n = distance(first, last);
```
--n;
while(n > 0) {
    typename iterator_traits<BidirectionalIterator>::value_type
    tmp = *first;
    *first++ = ***last;
    *last = tmp;
    n -= 2;
}

—end example]

27.4.2 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: input_iterator_tag, output_iterator_tag, forward_iterator_tag, bidirectional_iterator_tag and random_access_iterator_tag. For every iterator of type Iterator, iterator_traits<Iterator>::iterator_category shall be defined to be the most specific category tag that describes the iterator’s behavior.

namespace std {
    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 { };  
}

[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]

§ 27.4.2
27.4.3 Iterator operations

Since only random access iterators provide + and - operators, the library provides two function templates advance and distance. These function templates use + and - for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

```cpp
template<class InputIterator, class Distance>
constexpr void advance(InputIterator& i, Distance n);

Requires: n shall be negative only for bidirectional and random access iterators.

Effects: Increments (or decrements for negative n) iterator reference i by n.
```

```cpp
template<class InputIterator, class Distance>
constexpr typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Effects: If InputIterator meets the requirements of random access iterator, returns (last - first); otherwise, returns the number of increments needed to get from first to last.

```cpp
Requires: If InputIterator meets the requirements of random access iterator, last shall be reachable from first or first shall be reachable from last; otherwise, last shall be reachable from first.
```

```cpp
template<class InputIterator>
constexpr InputIterator next(InputIterator x, typename iterator_traits<InputIterator>::difference_type n = 1);
```

Effects: Equivalent to: advance(x, n); return x;

```cpp
template<class BidirectionalIterator>
constexpr BidirectionalIterator prev(BidirectionalIterator x, typename iterator_traits<BidirectionalIterator>::difference_type n = 1);
```

Effects: Equivalent to: advance(x, -n); return x;

27.5 Iterator adaptors

27.5.1 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. The fundamental relation between a reverse iterator and its corresponding iterator i is established by the identity: \&*(reverse_iterator(i)) == \&*(i - 1).

27.5.1.1 Class template reverse_iterator

```cpp
namespace std {
    template<class Iterator>
    class reverse_iterator {
        public:
            using iterator_type = Iterator;
            using iterator_category = typename iterator_traits<Iterator>::iterator_category;
            using value_type = typename iterator_traits<Iterator>::value_type;
            using difference_type = typename iterator_traits<Iterator>::difference_type;
            using pointer = typename iterator_traits<Iterator>::pointer;
            using reference = typename iterator_traits<Iterator>::reference;

            constexpr reverse_iterator();
            constexpr explicit reverse_iterator(Iterator x);
            template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);
            template<class U> constexpr reverse_iterator& operator=(const reverse_iterator<U>& u);

            constexpr Iterator base() const; // explicit
            constexpr reference operator*() const;
            constexpr pointer operator->() const;

            constexpr reverse_iterator& operator++();
            constexpr reverse_iterator operator++(int);
            constexpr reverse_iterator& operator--();
```
constexpr reverse_iterator operator--(int);
constexpr reverse_iterator operator+ (difference_type n) const;
constexpr reverse_iterator& operator+=(difference_type n);
constexpr reverse_iterator operator- (difference_type n) const;
constexpr reverse_iterator& operator-=(difference_type n);
constexpr
unspecified
operator[](difference_type n) const;

protected:
Iterator current;
};

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<(n
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator!=(n
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator>(n
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator>=(n
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr bool operator<=(n
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);

template<class Iterator1, class Iterator2>
constexpr auto operator-(n
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.base() - x.base());

template<class Iterator>
constexpr reverse_iterator<Iterator> operator+ (typename reverse_iterator<Iterator>::difference_type n,
    const reverse_iterator<Iterator>& x);

template<class Iterator>
constexpr reverse_iterator<Iterator> operator-(
    typename reverse_iterator<Iterator>::difference_type n,
    const reverse_iterator<Iterator>& x);

27.5.1.2 reverse_iterator requirements

The template parameter Iterator shall meet all the requirements of a Bidirectional Iterator (27.2.6).

Additionally, Iterator shall meet the requirements of a random access iterator (27.2.7) if any of the
members operator+ (27.5.1.3.8), operator- (27.5.1.3.10), operator+= (27.5.1.3.9), operator-= (27.5.1.3.11),
operator[] (27.5.1.3.12), or the non-member operators operator< (27.5.1.3.14), operator> (27.5.1.3.16),
or operator<= (27.5.1.3.18), operator>= (27.5.1.3.17) or operator+ (27.5.1.3.20) are
referenced in a way that requires instantiation (17.8.1).

27.5.1.3 reverse_iterator operations

27.5.1.3.1 reverse_iterator constructor

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.

```cpp
constexpr explicit reverse_iterator(Iterator x);
```

Effects: Initializes current with x.

```cpp
template<class U> constexpr reverse_iterator(const reverse_iterator<U>& u);
```

Effects: Initializes current with u.current.

27.5.1.3.2 reverse_iterator::operator= [reverse.iter.op=]

```cpp
template<class U>
constexpr reverse_iterator& operator=(const reverse_iterator<U>& u);
```

Effects: Assigns u.base() to current.

Returns: *this.

27.5.1.3.3 Conversion [reverse.iter.conv]

```cpp
constexpr Iterator base() const; // explicit
```

Returns: current.

27.5.1.3.4 operator* [reverse.iter.op.star]

```cpp
constexpr reference operator*() const;
```

Effects: As if by:

```
Iterator tmp = current;
return *--tmp;
```

27.5.1.3.5 operator-> [reverse.iter.opref]

```cpp
constexpr pointer operator->() const;
```

Returns: addressof(operator*()).

27.5.1.3.6 operator++ [reverse.iter.op++]

```cpp
constexpr reverse_iterator& operator++();
```

Effects: As if by: --current;

Returns: *this.

```cpp
constexpr reverse_iterator operator++(int);
```

Effects: As if by:

```
reverse_iterator tmp = *this;
--current;
return tmp;
```

27.5.1.3.7 operator-- [reverse.iter.op--]

```cpp
constexpr reverse_iterator& operator--();
```

Effects: As if by ++current.

Returns: *this.

```cpp
constexpr reverse_iterator operator--(int);
```

Effects: As if by:

```
reverse_iterator tmp = *this;
++current;
return tmp;
```
27.5.1.3.8 operator+

constexpr reverse_iterator operator+(difference_type n) const;

Returns: reverse_iterator(current-n).

27.5.1.3.9 operator++

constexpr reverse_iterator& operator+=(difference_type n);

Effects: As if by: current -= n;

Returns: *this.

27.5.1.3.10 operator-

constexpr reverse_iterator operator-(difference_type n) const;

Returns: reverse_iterator(current+n).

27.5.1.3.11 operator--

constexpr reverse_iterator& operator--(difference_type n);

Effects: As if by: current += n;

Returns: *this.

27.5.1.3.12 operator[]

constexpr unspecified operator[](difference_type n) const;

Returns: current[-n-1].

27.5.1.3.13 operator==

template<class Iterator1, class Iterator2>
constexpr bool operator==(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Returns: x.current == y.current.

27.5.1.3.14 operator<

template<class Iterator1, class Iterator2>
constexpr bool operator<(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Returns: x.current > y.current.

27.5.1.3.15 operator!=

template<class Iterator1, class Iterator2>
constexpr bool operator!=(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Returns: x.current != y.current.

27.5.1.3.16 operator>

template<class Iterator1, class Iterator2>
constexpr bool operator>(const reverse_iterator<Iterator1>& x, const reverse_iterator<Iterator2>& y);

Returns: x.current < y.current.
27.5.1.3.17  operator\>=

\[
\text{template<class Iterator1, class Iterator2>}
\text{constexpr bool operator\>=(}
\text{const reverse_iterator<Iterator1>& x,}
\text{const reverse_iterator<Iterator2>& y);}
\]

1  Returns: \(x.\text{current} \leq y.\text{current}\).

27.5.1.3.18  operator\<=

\[
\text{template<class Iterator1, class Iterator2>}
\text{constexpr bool operator\<=(}
\text{const reverse_iterator<Iterator1>& x,}
\text{const reverse_iterator<Iterator2>& y);}
\]

1  Returns: \(x.\text{current} \geq y.\text{current}\).

27.5.1.3.19  operator- [reverse.iter.opdiff]

\[
\text{template<class Iterator1, class Iterator2>}
\text{constexpr auto operator-}
\text{(const reverse_iterator<Iterator1>& x,}
\text{const reverse_iterator<Iterator2>& y) \rightarrow decltype(y.\text{base()} - x.\text{base()});}
\]

1  Returns: \(y.\text{current} - x.\text{current}\).

27.5.1.3.20  operator+ [reverse.iter.opsum]

\[
\text{template<class Iterator>}
\text{constexpr reverse_iterator<Iterator> operator+}
\text{(typename reverse_iterator<Iterator>::difference_type n,}
\text{const reverse_iterator<Iterator>& x);}\]

1  Returns: \(\text{reverse_iterator<Iterator> (x.\text{current} - n)}\).

27.5.1.3.21  Non-member function \text{make_reverse_iterator()}

\[
\text{template<class Iterator>}
\text{constexpr reverse_iterator<Iterator> make_reverse_iterator(Iterator i);}\]

1  Returns: \(\text{reverse_iterator<Iterator>(i)}\).

27.5.2  Insert iterators [insert.iterators]

1  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, 
\[
\text{while (first != last) *result++ = *first++;}
\]

causes a range \([\text{first, last})\) to be copied into a range starting with result. The same code with \text{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.

2  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. \text{operator}\* returns the insert iterator itself. The assignment \text{operator=}\((\text{const T& x})\) is defined on insert iterators to allow writing into them, it inserts \(x\) right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. \text{back_insert_iterator} inserts elements at the beginning of a container, and \text{insert_iterator} inserts elements where the iterator points to in a container. \text{back_inserter}, \text{front_inserter}, and \text{inserter} are three functions making the insert iterators out of a container.

27.5.2.1  Class template \text{back_insert_iterator}  [back.insert.iterator]

\[
\text{namespace std \{}\]
\text{class back_insert_iterator \{}\]
\text{protected:\}
\text{Container* container;\]

§ 27.5.2.1
public:
    using iterator_category = output_iterator_tag;
    using value_type = void;
    using difference_type = void;
    using pointer = void;
    using reference = void;
    using container_type = Container;

    explicit back_insert_iterator(Container& x);
    back_insert_iterator& operator=(const typename Container::value_type& value);
    back_insert_iterator& operator=(typename Container::value_type&& value);
    back_insert_iterator& operator*();
    back_insert_iterator& operator++();
    back_insert_iterator operator++(int);
};

template<class Container>
    back_insert_iterator<Container> back_inserter(Container& x);

27.5.2.2 back_insert_iterator operations [back.insert.iter.ops]
27.5.2.2.1 back_insert_iterator constructor [back.insert.iter.cons]
    explicit back_insert_iterator(Container& x);
1 Effects: Initializes container with addressof(x).

27.5.2.2.2 back_insert_iterator::operator= [back.insert.iter.op=]
    back_insert_iterator& operator=(const typename Container::value_type& value);
1 Effects: As if by: container->push_back(value);
2 Returns: *this.

    back_insert_iterator& operator=(typename Container::value_type&& value);
3 Effects: As if by: container->push_back(std::move(value));
4 Returns: *this.

27.5.2.2.3 back_insert_iterator::operator* [back.insert.iter.op*]
    back_insert_iterator& operator*();
1 Returns: *this.

27.5.2.2.4 back_insert_iterator::operator++ [back.insert.iter.op++]
    back_insert_iterator& operator++();
    back_insert_iterator operator++(int);
1 Returns: *this.

27.5.2.2.5 back_inserter [back.inserter]
    template<class Container>
        back_insert_iterator<Container> back_inserter(Container& x);
1 Returns: back_insert_iterator<Container>(x).

27.5.2.3 Class template front_insert_iterator [front.insert.iterator]
    namespace std {
        template<class Container>
            class front_insert_iterator {
                protected:
                    Container* container;

§ 27.5.2.3
public:
    using iterator_category = output_iterator_tag;
    using value_type = void;
    using difference_type = void;
    using pointer = void;
    using reference = void;
    using container_type = Container;

    explicit front_insert_iterator(Container& x);
    front_insert_iterator& operator=(const typename Container::value_type& value);
    front_insert_iterator& operator=(typename Container::value_type&& value);
    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);
};

template<class Container>
    front_insert_iterator<Container> front_inserter(Container& x);

27.5.2.4  front_insert_iterator operations  
27.5.2.4.1  front_insert_iterator constructor

explicit front_insert_iterator(Container& x);

1 Effects: Initializes container with addressof(x).

27.5.2.4.2  front_insert_iterator::operator=

front_insert_iterator& operator=(const typename Container::value_type& value);

1 Effects: As if by: container->push_front(value);
2 Returns: *this.

front_insert_iterator& operator=(typename Container::value_type&& value);

3 Effects: As if by: container->push_front(std::move(value));
4 Returns: *this.

27.5.2.4.3  front_insert_iterator::operator*

front_insert_iterator& operator*();

1 Returns: *this.

27.5.2.4.4  front_insert_iterator::operator++

front_insert_iterator& operator++();
front_insert_iterator operator++(int);

1 Returns: *this.

27.5.2.4.5  front_inserter

template<class Container>
    front_insert_iterator<Container> front_inserter(Container& x);

1 Returns: front_insert_iterator<Container>(x).

27.5.2.5  Class template insert_iterator

namespace std {
    template<class Container>
    class insert_iterator {
    protected:
        Container* container;
        typename Container::iterator iter;
};
public:
using iterator_category = output_iterator_tag;
using value_type = void;
using difference_type = void;
using pointer = void;
using reference = void;
using container_type = Container;

insert_iterator(Container& x, typename Container::iterator i);
insert_iterator& operator=(const typename Container::value_type& value);
insert_iterator& operator=(typename Container::value_type&& value);

insert_iterator& operator*();
insert_iterator& operator++();
insert_iterator& operator++(int);
};

template<class Container>
insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

27.5.2.6 insert_iterator operations

27.5.2.6.1 insert_iterator constructor

insert_iterator(Container& x, typename Container::iterator i);

Effects: Initializes container with addressof(x) and iter with i.

27.5.2.6.2 insert_iterator::operator=

insert_iterator& operator=(const typename Container::value_type& value);

Effects: As if by:
iter = container->insert(iter, value);
++iter;

Returns: *this.

insert_iterator& operator=(typename Container::value_type&& value);

Effects: As if by:
iter = container->insert(iter, std::move(value));
++iter;

Returns: *this.

27.5.2.6.3 insert_iterator::operator*

insert_iterator& operator*();

Returns: *this.

27.5.2.6.4 insert_iterator::operator++

insert_iterator& operator++();
insert_iterator& operator++(int);

Returns: *this.

27.5.2.6.5 inserter

template<class Container>
insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

Returns: insert_iterator<Container>(x, i).
27.5.3 Move iterators

1 Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its indirection operator implicitly converts the value returned by the underlying iterator’s indirection operator to an rvalue. Some generic algorithms can be called with move iterators to replace copying with moving.

2 [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]

27.5.3.1 Class template `move_iterator`

namespace std {
    template<class Iterator>
    class move_iterator {
    public:
        using iterator_type = Iterator;
        using iterator_category = typename iterator_traits<Iterator>::iterator_category;
        using value_type = typename iterator_traits<Iterator>::value_type;
        using difference_type = typename iterator_traits<Iterator>::difference_type;
        using pointer = Iterator;
        using reference = see below;

        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 pointer operator->() const;

        constexpr move_iterator& operator++();
        constexpr move_iterator 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 unspecified operator[](difference_type n) const;

    private:
        Iterator current;  // exposition only
    };

    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 bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

§ 27.5.3.1 873
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 Iterator>  
constexpr move_iterator<Iterator> operator+  
    (typename move_iterator<Iterator>::difference_type n,  
     const move_iterator<Iterator>& x);  

template<class Iterator>  
constexpr move_iterator<Iterator> make_move_iterator(Iterator i);  

1 Let \( R \) denote \( \text{iterator_traits<Iterator>::reference} \). If \( \text{is\_reference\_v<R> \text{is true} } \), the template specialization \( \text{move\_iterator<Iterator>} \) shall define the nested type named \( \text{reference} \) as a synonym for \( \text{remove\_reference\_t<R>&&} \), otherwise as a synonym for \( R \).

27.5.3.2 move_iterator requirements [move.iter.requirements]  
The template parameter \( \text{Iterator} \) shall meet the requirements of an input iterator (27.2.3). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator (27.2.6) or a Random Access Iterator (27.2.7), respectively.

27.5.3.3 move_iterator operations [move.iter.ops]  
27.5.3.3.1 move_iterator constructors [move.iter.op.const]

constexpr move_iterator();  
   
Effects: Constructs a \( \text{move\_iterator} \), value-initializing \( \text{current} \). 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 \( \text{Iterator} \).

constexpr explicit move_iterator(Iterator i);  
   
Effects: Constructs a \( \text{move\_iterator} \), initializing \( \text{current} \) with \( i \).

template<U> constexpr move_iterator(const move_iterator<U>& u);  
   
Effects: Constructs a \( \text{move\_iterator} \), initializing \( \text{current} \) with \( u.\text{base()} \).

Requires: \( U \) shall be convertible to \( \text{Iterator} \).

27.5.3.3.2 move_iterator::operator= [move.iter.op=]  

template<U> constexpr move_iterator& operator=(const move_iterator<U>& u);  
   
Effects: Assigns \( u.\text{base()} \) to \( \text{current} \).

Requires: \( U \) shall be convertible to \( \text{Iterator} \).

27.5.3.3.3 move_iterator conversion [move.iter.op.conv]  

constexpr Iterator base() const;  
   
Returns: \( \text{current} \).

27.5.3.3.4 move_iterator::operator* [move.iter.op.star]  

constexpr reference operator*() const;  
   
Returns: \text{static\_cast<reference>(*current)}.

27.5.3.3.5 move_iterator::operator-> [move.iter.op.ref]  

constexpr pointer operator->() const;  
   
Returns: \( \text{current} \).

§ 27.5.3.3.5  874
move_iterator::operator++

```cpp
constexpr move_iterator& operator++();
```

1. **Effects:** As if by `++current`.
2. **Returns:** *this.

```cpp
constexpr move_iterator operator++(int);
```

1. **Effects:** As if by:
   ```cpp
   move_iterator tmp = *this;
   ++current;
   return tmp;
   ```

27.5.3.3.7 move_iterator::operator--

```cpp
constexpr move_iterator& operator--();
```

1. **Effects:** As if by `--current`.
2. **Returns:** *this.

```cpp
constexpr move_iterator operator--(int);
```

1. **Effects:** As if by:
   ```cpp
   move_iterator tmp = *this;
   --current;
   return tmp;
   ```

27.5.3.3.8 move_iterator::operator+

```cpp
constexpr move_iterator operator+(difference_type n) const;
```

1. **Returns:** move_iterator(current + n).

27.5.3.3.9 move_iterator::operator+=

```cpp
constexpr move_iterator& operator+=(difference_type n);
```

1. **Effects:** As if by: `current += n`;
2. **Returns:** *this.

27.5.3.3.10 move_iterator::operator-

```cpp
constexpr move_iterator operator-(difference_type n) const;
```

1. **Returns:** move_iterator(current - n).

27.5.3.3.11 move_iterator::operator-=

```cpp
constexpr move_iterator& operator-=(difference_type n);
```

1. **Effects:** As if by: `current -= n`;
2. **Returns:** *this.

27.5.3.3.12 move_iterator::operator[]

```cpp
constexpr unspecified operator[](difference_type n) const;
```

1. **Returns:** std::move(current[n]).

27.5.3.3.13 move_iterator comparisons

```cpp
template<class Iterator1, class Iterator2>
constexpr bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

1. **Returns:** `x.base() == y.base()`.

```cpp
template<class Iterator1, class Iterator2>
constexpr bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

2. **Returns:** `(x != y)`.

§ 27.5.3.3.13
template<class Iterator1, class Iterator2>
constexpr bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: x.base() < y.base().

template<class Iterator1, class Iterator2>
constexpr bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: !(y < x).

template<class Iterator1, class Iterator2>
constexpr bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: y < x.

template<class Iterator1, class Iterator2>
constexpr bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

Returns: !(x < y).

27.5.3.3.14 move_iterator 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());

Returns: x.base() - y.base().

template<class Iterator>
constexpr move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);

Returns: x + n.

template<class Iterator>
constexpr move_iterator<Iterator> make_move_iterator(Iterator i);

Returns: move_iterator<Iterator>(i).

27.6 Stream iterators

To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[Example:

```cpp
partial_sum(istream_iterator<double, char>(cin),
            istream_iterator<double, char>(),
            ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating-point numbers from cin, and prints the partial sums onto cout. —end example]

27.6.1 Class template istream_iterator

The class template istream_iterator is an input iterator (27.2.3) that reads (using operator>>) successive elements from the input stream for which it was constructed. After it is constructed, and every time ++ is used, the iterator reads and stores a value of T. If the iterator fails to read and store a value of T (fail() on the stream returns true), the iterator becomes equal to the end-of-stream iterator value. The constructor with no arguments istream_iterator() always constructs an end-of-stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-stream iterator is not defined. For any other iterator value a const T& is returned. The result of operator-> on an end-of-stream iterator is not defined. For any other iterator value a const T* is returned. The behavior of a program that applies operator++() to an end-of-stream iterator is undefined. It is impossible to store things into istream iterators. The type T shall meet the DefaultConstructible, CopyConstructible, and CopyAssignable requirements.

Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.
namespace std {
  template<class T, class charT = char, class traits = char_traits<charT>,
           class Distance = pptrdiff_t>
  class istream_iterator {
    public:
      using iterator_category = input_iterator_tag;
      using value_type = T;
      using difference_type = Distance;
      using pointer = const T*;
      using reference = const T&;
      using char_type = charT;
      using traits_type = traits;
      using istream_type = basic_istream<charT,traits>;
      constexpr istream_iterator();
      istream_iterator(istream_type& s);
      istream_iterator(const istream_iterator& x) = default;
      ~istream_iterator() = default;
      const T& operator*() const;
      const T* operator->() const;
      istream_iterator& operator++();
      istream_iterator operator++(int);

    private:
      basic_istream<charT,traits>* in_stream;  // exposition only
      T value;                                // exposition only
  };

  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);

  § 27.6.1.1 istream_iterator constructors and destructor

  constexpr istream_iterator();
    Effects: Constructs the end-of-stream iterator. If is_trivially_default_constructible_v<T> is true, then this constructor is a constexpr constructor.

  istream_iterator(istream_type& s);
    Effects: Initializes in_stream with addressof(s). value may be initialized during construction or the first time it is referenced.

  istream_iterator(const istream_iterator& x) = default;
    Effects: Constructs a copy of x. If is_trivially_copy_constructible_v<T> is true, then this constructor is a trivial copy constructor.

  ~istream_iterator() = default;
    Effects: The iterator is destroyed. If is_trivially_destructible_v<T> is true, then this destructor is a trivial destructor.
27.6.1.2 istream_iterator operations [istream.iterator.ops]

const T& operator*() const;
Returns: value.

const T* operator->() const;
Returns: addressof(operator*()).

istream_iterator& operator++();
Requires: in_stream != 0.
Effects: As if by: *in_stream >> value;
Returns: *this.

istream_iterator operator++(int);
Requires: in_stream != 0.
Effects: As if by:

istream_iterator tmp = *this;
*in_stream >> value;
return (tmp);

template<class T, class charT, class traits, class Distance>
bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
const istream_iterator<T,charT,traits,Distance>& y);
Returns: x.in_stream == y.in_stream.

template<class T, class charT, class traits, class Distance>
bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
const istream_iterator<T,charT,traits,Distance>& y);
Returns: !(x == y)

27.6.2 Class template ostream_iterator [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. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

while (first != last)
*result++ = *first++;

ostream_iterator is defined as:

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 = void;
using pointer = void;
using reference = void;
using char_type = charT;
using traits_type = traits;
using ostream_type = basic_ostream<charT,traits>;

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 T& value);
27.6.2.1 ostream_iterator constructors and destructor

ostream_iterator(ostream_type& s);
1 Effects: Initializes out_stream with addressof(s) and delim with null.

ostream_iterator(ostream_type& s, const charT* delimiter);
2 Effects: Initializes out_stream with addressof(s) and delim with delimiter.

ostream_iterator(const ostream_iterator& x);
3 Effects: Constructs a copy of x.

~ostream_iterator();
4 Effects: The iterator is destroyed.

27.6.2.2 ostream_iterator operations

ostream_iterator& operator=(const T& value);
1 Effects: As if by:
   *out_stream << value;
   if (delim != 0)
      *out_stream << delim;
   return *this;

ostream_iterator& operator*();
2 Returns: *this.

ostream_iterator& operator++();
3 Returns: *this.

27.6.3 Class template istreambuf_iterator

The class template istreambuf_iterator defines an input iterator (27.2.3) that reads successive characters from the streambuf for which it was constructed. operator* provides access to the current input character, if any. Each time operator++ is evaluated, the iterator advances to the next input character. If the end of stream is reached (streambuf_type::sgetc() returns traits::eof()), the iterator becomes equal to the end-of-stream iterator value. The default constructor istreambuf_iterator() and the constructor istreambuf_iterator(0) 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;
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;
charT operator*() const;
proxy operator++(int);
bool equal(const istreambuf_iterator& b) const;

private:
  streambuf_type* sbuf_;  // exposition only
);

template<class charT, class traits>
bool operator==(const istreambuf_iterator<charT,traits>& a,
               const istreambuf_iterator<charT,traits>& b);

namespace std {
  template<class charT, class traits = char_traits<charT>>
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_; }
  }
};

1 Class istreambuf_iterator<charT, traits>::proxy is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class istreambuf_iterator<charT, traits>::proxy provides a temporary placeholder as the return value of the post-increment operator (operator++). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

27.6.3.2 istreambuf_iterator constructors

1 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.

istreambuf_iterator(istream_type& s) noexcept;

Effects: Initializes sbuf_ with s.rdbuf().

istreambuf_iterator(streambuf_type* s) noexcept;

Effects: Initializes sbuf_ with s.

§ 27.6.3.2
istreambuf_iterator(const proxy& p) noexcept;

**Effects:** Initializes sbuf_ with p.sbuf_.

### 27.6.3.3 istreambuf_iterator 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).

**template<class charT, class traits>**

**bool operator!=(const istreambuf_iterator<charT,traits>& a, const istreambuf_iterator<charT,traits>& b);**

**Returns:** !a.equal(b).

### 27.6.4 Class template ostreambuf_iterator

**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 = void;

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>;

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_;     // exposition only

};

**}**

1. The class template `ostreambuf_iterator` writes successive `characters` onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.
27.6.4.1 ostreambuf_iterator constructors

\begin{verbatim}
ostreambuf_iterator(ostream_type& s) noexcept;
\end{verbatim}
\begin{itemize}
\item Requires: \texttt{s.rdbuf()} shall not be a null pointer.
\item Effects: Initializes \texttt{sbuf_} with \texttt{s.rdbuf()}.
\end{itemize}

\begin{verbatim}
ostreambuf_iterator(streambuf_type* s) noexcept;
\end{verbatim}
\begin{itemize}
\item Requires: \texttt{s} shall not be a null pointer.
\item Effects: Initializes \texttt{sbuf_} with \texttt{s}.
\end{itemize}

27.6.4.2 ostreambuf_iterator operations

\begin{verbatim}
ostreambuf_iterator& operator=(charT c);
\end{verbatim}
\begin{itemize}
\item Effects: If \texttt{failed()} yields \texttt{false}, calls \texttt{sbuf_->sputc(c)}; otherwise has no effect.
\item Returns: \texttt{*this}.
\end{itemize}

\begin{verbatim}
ostreambuf_iterator& operator*();
\end{verbatim}
\begin{itemize}
\item Returns: \texttt{*this}.
\end{itemize}

\begin{verbatim}
ostreambuf_iterator& operator++();
ostreambuf_iterator& operator++(int);
\end{verbatim}
\begin{itemize}
\item Returns: \texttt{*this}.
\end{itemize}

\begin{verbatim}
bool failed() const noexcept;
\end{verbatim}
\begin{itemize}
\item Returns: \texttt{true} if in any prior use of member \texttt{operator=}, the call to \texttt{sbuf_->sputc()} returned \texttt{traits::eof()}; or \texttt{false} otherwise.
\end{itemize}

27.7 Range access

\begin{verbatim}
template<class C> constexpr auto begin(C& c) -> decltype(c.begin());
template<class C> constexpr auto begin(const C& c) -> decltype(c.begin());
\end{verbatim}
\begin{itemize}
\item Returns: \texttt{c.begin()}.\n\item Returns: \texttt{c.end()}.\n\item Returns: \texttt{array}.\n\item Returns: \texttt{array + N}.\n\item Returns: \texttt{std::begin(c)}.\n\item Returns: \texttt{std::end(c)}.\n\end{itemize}

\begin{verbatim}
template<class C> constexpr auto end(C& c) -> decltype(c.end());
template<class C> constexpr auto end(const C& c) -> decltype(c.end());
\end{verbatim}
\begin{itemize}
\item Returns: \texttt{c.end()}.\n\item Returns: \texttt{c.end()}.\n\item Returns: \texttt{std::begin(c)}.\n\item Returns: \texttt{std::end(c)}.\n\end{itemize}

\begin{verbatim}
template<class C> constexpr auto cbegin(const C& c) noexcept(noexcept(std::begin(c)))
-> decltype(std::begin(c));
template<class C> constexpr auto cend(const C& c) noexcept(noexcept(std::end(c)))
-> decltype(std::end(c));
\end{verbatim}
\begin{itemize}
\item Returns: \texttt{c.begin()}.\n\item Returns: \texttt{c.end()}.\n\end{itemize}
template<class C> constexpr auto rend(C& c) -> decltype(c.rend());
Returns: c.rend().

template<class T, size_t N> constexpr reverse_iterator<T*> rbegin(T (&array)[N]);
Returns: reverse_iterator<T*>(array + N).

template<class T, size_t N> constexpr reverse_iterator<T*> rend(T (&array)[N]);
Returns: reverse_iterator<T*>(array).

template<class E> constexpr reverse_iterator<const E*> rbegin(initializer_list<E> il);
Returns: reverse_iterator<const E*>(il.end()).

template<class E> constexpr reverse_iterator<const E*> rend(initializer_list<E> il);
Returns: reverse_iterator<const E*>(il.begin()).

template<class C> constexpr auto crbegin(const C& c) -> decltype(std::rbegin(c));
Returns: std::rbegin(c).

template<class C> constexpr auto crend(const C& c) -> decltype(std::rend(c));
Returns: std::rend(c).

27.8 Container access

In addition to being available via inclusion of the <iterator> header, the function templates in 27.8 are available when any of the following headers are included: <array>, <deque>, <forward_list>, <list>, <map>, <regex>, <set>, <string>, <unordered_map>, <unordered_set>, and <vector>.

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> [[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().
28 Algorithms library

28.1 General

This Clause describes components that C++ programs may use to perform algorithmic operations on containers (Clause 26) 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 100.

Table 100 — Algorithms library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.5</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>28.6</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>28.7</td>
<td>&lt;algorithm&gt;</td>
</tr>
<tr>
<td>28.8</td>
<td>&lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

28.2 Header <algorithm> synopsis

```cpp
#include <initializer_list>

namespace std {
    // 28.5, non-modifying sequence operations
    // 28.5.1, 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 28.4.5
                 ForwardIterator first, ForwardIterator last, Predicate pred);

    // 28.5.2, 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 28.4.5
                ForwardIterator first, ForwardIterator last, Predicate pred);

    // 28.5.3, 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 28.4.5
                 ForwardIterator first, ForwardIterator last, Predicate pred);

    // 28.5.4, for_each
    template<class InputIterator, class Function>
    constexpr Function for_each(InputIterator first, InputIterator last, Function f);
    template<class ExecutionPolicy, class ForwardIterator, class Function>
    void for_each(ExecutionPolicy&& exec, // see 28.4.5
                  ForwardIterator first, ForwardIterator last, Function f);
    template<class InputIterator, class Size, class Function>
    constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);
    template<class ExecutionPolicy, class ForwardIterator, Size, class Function>
    ForwardIterator for_each_n(ExecutionPolicy&& exec, // see 28.4.5
                                ForwardIterator first, Size n, Function f);
```

§ 28.2
// 28.5.5, find

```cpp
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 28.4.5
    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 28.4.5
    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 28.4.5
    ForwardIterator first, ForwardIterator last,
    Predicate pred);
```

// 28.5.6, find end

```cpp
template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr ForwardIterator1
    find_end(ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 28.4.5
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
    class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
    find_end(ExecutionPolicy&& exec, // see 28.4.5
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

// 28.5.7, find first

```cpp
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 28.4.5
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1,
    class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
    find_first_of(ExecutionPolicy&& exec, // see 28.4.5
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```
find_first_of(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

// 28.5.8, 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 28.4.5
ForwardIterator first, ForwardIterator last);
template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
ForwardIterator
adjacent_find(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last,
BinaryPredicate pred);

// 28.5.9, 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 28.4.5
ForwardIterator first, ForwardIterator last, const T& value);
template<class InputIterator, class Predicate>
constexpr typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
constexpr typename iterator_traits<ForwardIterator>::difference_type
count_if(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last, Predicate pred);

// 28.5.10, 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 InputIterator1, class InputIterator2>
constexpr pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 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>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
pair<ForwardIterator1, ForwardIterator2>
mismatch(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

// 28.5.11, equal
template<class InputIterator1, class InputIterator2>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2);

// 28.5.12, is permutation
template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2,
                              BinaryPredicate pred);

// 28.5.13, 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);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
search(ExecutionPolicy&& exec, // see 28.4.5
      ForwardIterator1 first1, ForwardIterator1 last1,
      ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1
search(ExecutionPolicy&& exec, // see 28.4.5
      ForwardIterator1 first1, ForwardIterator1 last1,
      ForwardIterator2 first2, ForwardIterator2 last2,
      BinaryPredicate pred);

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 28.4.5
         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 28.4.5
         ForwardIterator first, ForwardIterator last,
         Size count, const T& value,
         BinaryPredicate pred);

template<class ForwardIterator, class Searcher>
constexpr ForwardIterator
search(ForwardIterator first, ForwardIterator last, const Searcher& searcher);

// 28.6, mutating sequence operations
// 28.6.1, copy
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, // see 28.4.5
                       ForwardIterator1 first, ForwardIterator1 last,
                       ForwardIterator2 result);
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, // see 28.4.5
ForwardIterator1 first, Size n,
ForwardIterator2 result);

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 28.4.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, Predicate pred);

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
BidirectionalIterator2 result);

// 28.6.2, 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 28.4.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result);

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
BidirectionalIterator2 result);

// 28.6.3, swap
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

// 28.6.4, transform
template<class InputIterator, class OutputIterator, class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first, InputIterator last,
OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2, class OutputIterator,
class BinaryOperation>
constexpr OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, OutputIterator result,
BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class UnaryOperation>
ForwardIterator2
transform(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, UnaryOperation op);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class ForwardIterator, class BinaryOperation>
ForwardIterator
transform(ExecutionPolicy&& exec, // see 28.4.5
     ForwardIterator1 first1, ForwardIterator1 last1,
     ForwardIterator2 first2, ForwardIterator result,
     BinaryOperation binary_op);

// 28.6.5, 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 28.4.5
     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 28.4.5
     ForwardIterator first, ForwardIterator last,
     Predicate pred, const T& new_value);
template<class InputIterator, class OutputIterator, class T>
constexpr OutputIterator replace_copy(InputIterator first, InputIterator last,
     OutputIterator result,
     const T& old_value, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2 replace_copy(ExecutionPolicy&& exec, // see 28.4.5
     ForwardIterator1 first, ForwardIterator1 last,
     ForwardIterator2 result,
     const T& old_value, const T& new_value);
template<class InputIterator, class OutputIterator, class Predicate, class T>
constexpr OutputIterator replace_copy_if(InputIterator first, InputIterator last,
     OutputIterator result,
     Predicate pred, const T& new_value);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Predicate, class T>
ForwardIterator2 replace_copy_if(ExecutionPolicy&& exec, // see 28.4.5
     ForwardIterator1 first, ForwardIterator1 last,
     ForwardIterator2 result,
     Predicate pred, const T& new_value);

// 28.6.6, 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, // see 28.4.5
     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, // see 28.4.5
     ForwardIterator first, Size n, const T& value);

// 28.6.7, generate
template<class ForwardIterator, class Generator>
constexpr void generate(ForwardIterator first, ForwardIterator last, Generator gen);
template<class ExecutionPolicy, class ForwardIterator, class Generator>
void generate(ExecutionPolicy&& exec, // see 28.4.5
     ForwardIterator first, ForwardIterator last, Generator gen);
template<class OutputIterator, class Size, class Generator>
constexpr OutputIterator generate_n(OutputIterator first, Size n, Generator gen);

template<class ExecutionPolicy, class ForwardIterator, class Size, class Generator>
ForwardIterator generate_n(ExecutionPolicy& exec, // see 28.4.5
  ForwardIterator first, Size n, Generator gen);

// 28.6.8, remove
template<class ForwardIterator, class T>
constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
ForwardIterator remove(ExecutionPolicy& exec, // see 28.4.5
  ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last, const T& value);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator remove_if(ExecutionPolicy& exec, // see 28.4.5
  ForwardIterator first, ForwardIterator last, Predicate pred);

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 28.4.5
  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 28.4.5
  ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result, Predicate pred);

// 28.6.9, 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 28.4.5
  ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ExecutionPolicy& exec, // see 28.4.5
  ForwardIterator first, ForwardIterator last, BinaryPredicate pred);

template<class InputIterator, class OutputIterator>
constexpr OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
constexpr OutputIterator unique_copy(InputIterator first, InputIterator last, OutputIterator result, BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 28.4.5
        ForwardIterator1 first, ForwardIterator1 last,
        ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator2
    unique_copy(ExecutionPolicy&& exec, // see 28.4.5
        ForwardIterator1 first, ForwardIterator1 last,
        ForwardIterator2 result, BinaryPredicate pred);

// 28.6.10, reverse
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);

template<class ExecutionPolicy, class BidirectionalIterator>
void reverse(ExecutionPolicy&& exec, // see 28.4.5
    BidirectionalIterator first, BidirectionalIterator last);

const expr OutputIterator
    reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
        OutputIterator result);

template<class ExecutionPolicy, class BidirectionalIterator, class ForwardIterator>
ForwardIterator
    reverse_copy(ExecutionPolicy&& exec, // see 28.4.5
        BidirectionalIterator first, BidirectionalIterator last,
        ForwardIterator result);

// 28.6.11, rotate
template<class ForwardIterator>
ForwardIterator rotate(ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator rotate(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator first,
    ForwardIterator middle,
    ForwardIterator last);

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 28.4.5
        ForwardIterator1 first, ForwardIterator1 middle,
        ForwardIterator1 last, ForwardIterator2 result);

// 28.6.12, sample
template<class PopulationIterator, class SampleIterator,
    class Distance, class UniformRandomBitGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
    SampleIterator out, Distance n,
    UniformRandomBitGenerator&& g);

// 28.6.13, shuffle
template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first,
    RandomAccessIterator last,
    UniformRandomBitGenerator&& g);

// 28.7.4, 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 28.4.5
ForwardIterator first, ForwardIterator last, Predicate pred);

template<class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator first,
ForwardIterator last, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator partition(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last, Predicate pred);

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 28.4.5
BidirectionalIterator first, BidirectionalIterator last, Predicate pred);

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 28.4.5
ForwardIterator first, ForwardIterator last,
ForwardIterator1 out_true, ForwardIterator2 out_false,
Predicate pred);

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
partition_point(ForwardIterator first, ForwardIterator last,
Predicate pred);

§ 28.2 893
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void stable_sort(ExecutionPolicy&& exec, // see 28.4.5
RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

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 28.4.5
RandomAccessIterator first,
RandomAccessIterator middle,
RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void partial_sort(ExecutionPolicy&& exec, // see 28.4.5
RandomAccessIterator first,
RandomAccessIterator middle,
RandomAccessIterator last, Compare comp);

template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
RandomAccessIterator result_first,
RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
RandomAccessIterator result_first,
RandomAccessIterator result_last,
Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last,
RandomAccessIterator result_first,
RandomAccessIterator result_last);

template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
class Compare>
RandomAccessIterator
partial_sort_copy(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last,
RandomAccessIterator result_first,
RandomAccessIterator result_last,
Compare comp);

constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
Compare comp);

template<class ExecutionPolicy, class ForwardIterator>
bool is_sorted(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
bool is_sorted(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator first, ForwardIterator last,
Compare comp);

template<class ForwardIterator>
constexpr ForwardIterator
is_sorted_until(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>
ForwardIterator
is_sorted_until(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator
is_sorted_until(ExecutionPolicy&& exec, // see 28.4.5
                ForwardIterator first, ForwardIterator last,
                Compare comp);

// 28.7.2, Nth element
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last, Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator>
void nth_element(ExecutionPolicy&& exec, // see 28.4.5
                RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void nth_element(ExecutionPolicy&& exec, // see 28.4.5
                RandomAccessIterator first, RandomAccessIterator nth,
                RandomAccessIterator last, Compare comp);

// 28.7.3, 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);

template<class ForwardIterator, class T>
constexpr ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
            const T& value);

template<class ForwardIterator, class T, class Compare>
constexpr ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
            const T& value, Compare comp);

template<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<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);

// 28.7.5, merge
template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);
template<class InputIterator1, class InputIterator2, class OutputIterator,
class Compare>
constexpr OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
template<class ExecutionPolicy, class InputIterator1, class InputIterator2,
class OutputIterator, class Compare>
constexpr OutputIterator
merge(ExecutionPolicy&& exec,
InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result, Compare comp);
template<class ExecutionPolicy, class InputIterator1, class InputIterator2,
class OutputIterator, class Compare>
OutputIterator
merge(ExecutionPolicy&& exec,
InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);

template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last, Compare comp);
template<class ExecutionPolicy, class BidirectionalIterator>
void inplace_merge(ExecutionPolicy&& exec,
BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last);
template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
void inplace_merge(ExecutionPolicy&& exec,
BidirectionalIterator first,
BidirectionalIterator middle,
BidirectionalIterator last, Compare comp);

// 28.7.6, set operations
template<class InputIterator1, class InputIterator2>
constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);
template<class InputIterator1, class InputIterator2, class Compare>
constexpr bool includes(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, Compare comp);
template<class ExecutionPolicy, class InputIterator1, class InputIterator2,
class Compare>
bool includes(ExecutionPolicy&& exec,
InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
   class Compare>
bool includes(ExecutionPolicy&& exec, // see 28.4.5  
   ForwardIterator1 first1, ForwardIterator1 last1,  
   ForwardIterator2 first2, ForwardIterator2 last2,  
   Compare comp);

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, // see 28.4.5  
   ForwardIterator1 first1, ForwardIterator1 last1,  
   ForwardIterator2 first2, ForwardIterator2 last2,  
   ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
   class Compare>
ForwardIterator  
set_union(ExecutionPolicy&& exec, // see 28.4.5  
   ForwardIterator1 first1, ForwardIterator1 last1,  
   ForwardIterator2 first2, ForwardIterator2 last2,  
   Compare comp);

template<class InputIterator1, class InputIterator2, class OutputIterator>  
constexpr OutputIterator  
set_intersection(InputIterator1 first1, InputIterator1 last1,  
   InputIterator2 first2, InputIterator2 last2,  
   OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,  
   class Compare>
ForwardIterator  
set_intersection(ExecutionPolicy&& exec, // see 28.4.5  
   ForwardIterator1 first1, ForwardIterator1 last1,  
   ForwardIterator2 first2, ForwardIterator2 last2,  
   Compare comp);

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 28.4.5
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 28.4.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
ForwardIterator result, Compare comp);

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, // see 28.4.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
ForwardIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class ForwardIterator, class Compare>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec, // see 28.4.5
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
ForwardIterator result, Compare comp);

// 28.7.7, heap operations
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);

template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
Compare comp);
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

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>
  RandomAccessIterator is_heap_until(ExecutionPolicy&& exec, // see 28.4.5
                                       RandomAccessIterator first, RandomAccessIterator last);
template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
  RandomAccessIterator is_heap_until(ExecutionPolicy&& exec, // see 28.4.5
                                       RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// 28.7.8, 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);

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);

template<class ForwardIterator>
  constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
template<class ForwardIterator, class Compare>
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
    Compare comp);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator min_element(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator min_element(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator first, ForwardIterator last,
    Compare comp);

template<class ForwardIterator>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
    Compare comp);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator max_element(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator max_element(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator first, ForwardIterator last,
    Compare comp);

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 28.4.5
    ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator>
minmax_element(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator first, ForwardIterator last,
    Compare comp);

// 28.7.9, 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);

// 28.7.10, 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 28.4.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class Compare>
bool
lexicographical_compare(ExecutionPolicy&& exec, // see 28.4.5
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
Compare comp);

// 28.7.11, 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>;

template<class InputIterator1, class InputIterator2>
constexpr auto
lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
InputIterator2 b2, InputIterator2 e2);

// 28.7.12, permutations
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);

template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
BidirectionalIterator last, Compare comp);

28.3 Algorithms requirements [algorithms.requirements]

1 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

2 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.

3 Throughout this Clause, the names of template parameters are used to express type requirements.

   (3.1) If an algorithm’s template parameter is named InputIterator, InputIterator1, or InputIterator2, the template argument shall satisfy the requirements of an input iterator (27.2.3).

   (3.2) If an algorithm’s template parameter is named OutputIterator, OutputIterator1, or OutputIterator2, the template argument shall satisfy the requirements of an output iterator (27.2.4).

   (3.3) If an algorithm’s template parameter is named ForwardIterator, ForwardIterator1, or ForwardIterator2, the template argument shall satisfy the requirements of a forward iterator (27.2.5).

   (3.4) If an algorithm’s template parameter is named BidirectionalIterator, BidirectionalIterator1, or BidirectionalIterator2, the template argument shall satisfy the requirements of a bidirectional iterator (27.2.6).

   (3.5) If an algorithm’s template parameter is named RandomAccessIterator, RandomAccessIterator1, or RandomAccessIterator2, the template argument shall satisfy the requirements of a random-access iterator (27.2.7).

4 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 (27.2). [Note: This requirement does not affect arguments that are named OutputIterator, OutputIterator1, or OutputIterator2, because output iterators must always be mutable. —end note]
Both in-place and copying versions are provided for certain algorithms. When such a version is provided for `algorithm` it is called `algorithm_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

The `Predicate` parameter is used whenever an algorithm expects a function object (23.14) 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, it should work correctly in the construct `pred(*first)` contextually converted to `bool` (Clause 7). The function object `pred` shall not apply any non-constant function through the dereferenced iterator.

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, it should work correctly in the construct `binary_pred(*first1, *first2)` contextually converted to `bool` (Clause 7). `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` (Clause 7). `binary_pred` shall not apply any non-constant function through the dereferenced iterators.

[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>` (23.14.5), or some equivalent solution. —end note]

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.

In the description of the algorithms operators `+` and `-` are used for some of the iterator categories for which they do not have to be defined. In these cases the semantics of `a+n` is the same as that of

```cpp
X tmp = a;
advance(tmp, n);
return tmp;
```
and that of `b-a` is the same as of

```cpp
return distance(a, b);
```

### 28.4 Parallel algorithms

This subclause describes components that C++ programs may use to perform operations on containers and other sequences in parallel.

#### 28.4.1 Terms and definitions

A `parallel algorithm` is a function template listed in this document with a template parameter named `ExecutionPolicy`.

Parallel algorithms access objects indirectly accessible via their arguments by invoking the following functions:

1. All operations of the categories of the iterators that the algorithm is instantiated with.
2. Operations on those sequence elements that are required by its specification.
3. User-provided function objects to be applied during the execution of the algorithm, if required by the specification.
4. Operations on those function objects required by the specification. [Note: See 28.1. —end note]

These functions are herein called `element access functions`. [Example: The `sort` function may invoke the following element access functions:

1. Operations of the random-access iterator of the actual template argument (as per 27.2.7), as implied by the name of the template parameter `RandomAccessIterator`.
2. The `swap` function on the elements of the sequence (as per the preconditions specified in 28.7.1.1).]

---

265) 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`. 

---

§ 28.4.1
28.4.2 Requirements on user-provided function objects [algorithms.parallel.user]

Unless otherwise specified, function objects passed into parallel algorithms as objects of type `Predicate`, `BinaryPredicate`, `Compare`, `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, `BinaryOperation2`, and the operators used by the analogous overloads to these parallel algorithms that could be formed by the invocation with the specified default predicate or operation (where applicable) shall not directly or indirectly modify objects via their arguments, nor shall they rely on the identity of the provided objects.

28.4.3 Effect of execution policies on algorithm execution [algorithms.parallel.exec]

Parallel algorithms have template parameters named `ExecutionPolicy` (23.19) which describe the manner in which the execution of these algorithms may be parallelized and the manner in which they apply the element access functions.

1. If an object is modified by an element access function, the algorithm will perform no other unsynchronized accesses to that object. The modifying element access functions are those which are specified as modifying the object in the relevant concept. [Note: For example, `swap()`, `++`, `--`, `@=`, and assignments modify the object. For the assignment and `@=` operators, only the left argument is modified. —end note]

2. Unless otherwise stated, implementations may make arbitrary copies of elements (with type `T`) from sequences where `is_trivially_copy_constructible_v<T>` and `is_trivially_destructible_v<T>` are true. [Note: This implies that user-supplied function objects should not rely on object identity of arguments for such input sequences. Users for whom the object identity of the arguments to these function objects is important should consider using a wrapping iterator that returns a non-copied implementation object such as `reference_wrapper<T>` (23.14.5) or some equivalent solution. —end note]

3. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::sequenced_policy` all occur in the calling thread of execution. [Note: The invocations are not interleaved; see 6.8.1. —end note]

4. The invocations of element access functions in parallel algorithms invoked with an execution policy object of type `execution::parallel_policy` are permitted to execute in either the invoking thread of execution or in a thread of execution implicitly created by the library to support parallel algorithm execution. If the threads of execution created by `thread` (33.3.2) 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:

```cpp
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
});
```

The program above has a data race because of the unsynchronized access to the container `v`. —end example]

[Example:

```cpp
int a[] = {1,2};
std::atomic<int> x{0};
std::for_each(std::execution::par, std::begin(a), std::end(a), [&] (int) {
  x.fetch_add(1, std::memory_order::relaxed);
  // spin wait for another iteration to change the value of x
  while (x.load(std::memory_order::relaxed) == 1) {} // incorrect: assumes execution order
});
```

The above example depends on the order of execution of the iterations, and will not terminate if both iterations are executed sequentially on the same thread of execution. —end example]

[Example:

```cpp
int x = 0;
std::mutex m;
int a[] = {1,2};
```]
The above example synchronizes access to object \( x \) ensuring that it is incremented correctly. — end example]}

6 The invocations of element access functions in parallel algorithms invoked with an execution policy of type \texttt{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 interleave with one another. — end note] Since \texttt{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. Thus, the synchronization with \texttt{execution::parallel_unsequenced_policy} is restricted as follows: A standard library function is \texttt{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. Vectorization-unsafe standard library functions may not be invoked by user code called from \texttt{execution::parallel_unsequenced_policy} algorithms. [Note: Implementations must ensure that internal synchronization inside standard library functions does not prevent forward progress when those functions are executed by threads of execution with weakly parallel forward progress guarantees. — end note] [ Example:

```cpp
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par_unseq, std::begin(a), std::end(a), [&] (int) {
    std::lock_guard<mutex> guard(m); // incorrect: lock_guard constructor calls m.lock()
    ++x;
});
```

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] [ Note: The semantics of the \texttt{execution::parallel_policy} or the \texttt{execution::parallel_unsequenced_policy} invocation allow the implementation to fall back to sequential execution if the system cannot parallelize an algorithm invocation due to lack of resources. — end note]}

7 If an invocation of a parallel algorithm uses threads of execution implicitly created by the library, then the invoking thread of execution will either

(7.1) — temporarily block with forward progress guarantee delegation (6.8.2.2) on the completion of these library-managed threads of execution, or

(7.2) — eventually execute an element access function;

the thread of execution will continue to do so until the algorithm is finished. [Note: In blocking with forward progress guarantee delegation in this context, a thread of execution created by the library is considered to have finished execution as soon as it has finished the execution of the particular element access function that the invoking thread of execution logically depends on. — end note]

8 The semantics of parallel algorithms invoked with an execution policy object of implementation-defined type are implementation-defined.

28.4.4 Parallel algorithm exceptions [algorithms.parallel.exceptions]

1 During the execution of a parallel algorithm, if temporary memory resources are required for parallelization and none are available, the algorithm throws a \texttt{bad_alloc} exception.

2 During the execution of a parallel algorithm, if the invocation of an element access function exits via an uncaught exception, the behavior is determined by the \texttt{ExecutionPolicy}.

28.4.5 ExecutionPolicy algorithm overloads [algorithms.parallel.overloads]

1 Parallel algorithms are algorithm overloads. Each parallel algorithm overload has an additional template type parameter named \texttt{ExecutionPolicy}, which is the first template parameter. Additionally, each parallel
algorithm overload has an additional function parameter of type `ExecutionPolicy&&`, which is the first function parameter. [Note: Not all algorithms have parallel algorithm overloads. — end note]

2 Unless otherwise specified, the semantics of `ExecutionPolicy` algorithm overloads are identical to their overloads without.

3 Unless otherwise specified, the complexity requirements of `ExecutionPolicy` algorithm overloads are relaxed from the complexity requirements of the overloads without as follows: when the guarantee says “at most `expr`” or “exactly `expr`” and does not specify the number of assignments or swaps, and `expr` is not already expressed with $\mathcal{O}(\cdot)$ notation, the complexity of the algorithm shall be $\mathcal{O}(\text{expr})$.

4 Parallel algorithms shall not participate in overload resolution unless `is_execution_policy_v<decay_t<ExecutionPolicy>>` is true.

28.5 Non-modifying sequence operations [alg.nonmodifying]

28.5.1 All of [alg.all_of]

```cpp
template<class InputIterator, class Predicate>
constexpr bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

1 Returns: true if `[first, last)` is empty or if `pred(*i)` is true for every iterator `i` in the range `[first, last)`, and false otherwise.

2 Complexity: At most `last - first` applications of the predicate.

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool all_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Predicate pred);
```

1 Returns: true if `[first, last)` is empty or if `pred(*i)` is true for every iterator `i` in the range `[first, last)`, and false otherwise.

2 Complexity: At most `last - first` applications of the predicate.

28.5.2 Any of [alg.any_of]

```cpp
template<class InputIterator, class Predicate>
constexpr bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

1 Returns: false if `[first, last)` is empty or if there is no iterator `i` in the range `[first, last)` such that `pred(*i)` is true, and true otherwise.

2 Complexity: At most `last - first` applications of the predicate.

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Predicate pred);
```

1 Returns: false if `[first, last)` is empty or if there is no iterator `i` in the range `[first, last)` such that `pred(*i)` is true, and true otherwise.

2 Complexity: At most `last - first` applications of the predicate.

28.5.3 None of [alg.none_of]

```cpp
template<class InputIterator, class Predicate>
constexpr bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

1 Returns: true if `[first, last)` is empty or if `pred(*i)` is false for every iterator `i` in the range `[first, last)`, and false otherwise.

2 Complexity: At most `last - first` applications of the predicate.

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
bool none_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Predicate pred);
```

1 Returns: true if `[first, last)` is empty or if `pred(*i)` is false for every iterator `i` in the range `[first, last)`, and false otherwise.

2 Complexity: At most `last - first` applications of the predicate.

28.5.4 For each [alg.foreach]

```cpp
template<class InputIterator, class Function>
constexpr Function for_each(InputIterator first, InputIterator last, Function f);
```

1 Requires: `Function` shall meet the requirements of `MoveConstructible` (Table 23). [Note: `Function` need not meet the requirements of `CopyConstructible` (Table 24). — end note]

2 Effects: Applies `f` to the result of dereferencing every iterator in the range `[first, last)`, starting from `first` and proceeding to `last - 1`. [Note: If the type of `first` satisfies the requirements of a mutable iterator, `f` may apply non-constant functions through the dereferenced iterator. — end note]

3 Returns: `f`.

4 Complexity: Applies `f` exactly `last - first` times.

5 Remarks: If `f` returns a result, the result is ignored.
template<class ExecutionPolicy, class ForwardIterator, class Function>
void for_each(ExecutionPolicy&& exec,
            ForwardIterator first, ForwardIterator last,
            Function f);

6 \textit{Requires}: Function shall meet the requirements of CopyConstructible.

7 \textit{Effects}: Applies f to the result of dereferencing every iterator in the range \([\text{first}, \text{last})\). \textit{Note}: If the type of \textit{first} satisfies the requirements of a mutable iterator, \textit{f} may apply non-constant functions through the dereferenced iterator. \textit{end note}

8 \textit{Complexity}: Applies \textit{f} exactly \(\text{last} - \text{first}\) times.

9 \textit{Remarks}: If \textit{f} returns a result, the result is ignored. Implementations do not have the freedom granted under 28.4.3 to make arbitrary copies of elements from the input sequence.

[\textit{Note}: Does not return a copy of its \textit{Function} parameter, since parallelization may not permit efficient state accumulation. \textit{end note}]

template<class InputIterator, class Size, class Function>
constexpr InputIterator for_each_n(InputIterator first, Size n, Function f);

11 \textit{Requires}: Function shall meet the requirements of MoveConstructible \textit{Note}: Function need not meet the requirements of CopyConstructible. \textit{end note}

12 \textit{Requires}: \(n \geq 0\).

13 \textit{Effects}: Applies \textit{f} to the result of dereferencing every iterator in the range \([\text{first}, \text{first} + n)\) in order. \textit{Note}: If the type of \textit{first} satisfies the requirements of a mutable iterator, \textit{f} may apply non-constant functions through the dereferenced iterator. \textit{end note}

14 \textit{Returns}: \textit{first} + \textit{n}.

15 \textit{Remarks}: If \textit{f} returns a result, the result is ignored.

template<class ExecutionPolicy, class ForwardIterator, class Size, class Function>
ForwardIterator for_each_n(ExecutionPolicy&& exec, ForwardIterator first, Size n,
                            Function f);

16 \textit{Requires}: Function shall meet the requirements of CopyConstructible.

17 \textit{Requires}: \(n \geq 0\).

18 \textit{Effects}: Applies \textit{f} to the result of dereferencing every iterator in the range \([\text{first}, \text{first} + n)\). \textit{Note}: If the type of \textit{first} satisfies the requirements of a mutable iterator, \textit{f} may apply non-constant functions through the dereferenced iterator. \textit{end note}

19 \textit{Returns}: \textit{first} + \textit{n}.

20 \textit{Remarks}: If \textit{f} returns a result, the result is ignored. Implementations do not have the freedom granted under 28.4.3 to make arbitrary copies of elements from the input sequence.

28.5.5 \textbf{Find} \hfill \textit{[alg.find]}

template<class InputIterator, class T>
constexpr InputIterator find(InputIterator first, InputIterator last,
                             const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
ForwardIterator find(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                     const T& value);

template<class InputIterator, class Predicate>
constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                 Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator find_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                        Predicate pred);

template<class InputIterator, class Predicate>
constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                    Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
ForwardIterator find_if_not(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
Predicate pred);

Returns: The first iterator i in the range [first, last) for which the following corresponding
conditions hold: *i == value, pred(*i) != false, pred(*i) == false. Returns last if no such
iterator is found.

Complexity: At most last - first applications of the corresponding predicate.

28.5.6 Find end

template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_end(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
constexpr ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
BinaryPredicate pred);

Effects: Finds a subsequence of equal values in a sequence.

Returns: The last iterator i in the range [first1, last1 - (last2 - first2)) such that for every
non-negative integer n < (last2 - first2), the following corresponding conditions hold: *(i +
n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false. Returns last1 if [first2,
last2) is empty or if no such iterator is found.

Complexity: At most (last2 - first2) * (last1 - first1 - (last2 - first2) + 1) applica-
tions of the corresponding predicate.

28.5.7 Find first

template<class InputIterator, class ForwardIterator>
constexpr InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1
find_first_of(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator, class ForwardIterator,
class BinaryPredicate>
constexpr InputIterator
find_first_of(InputIterator first1, InputIterator last1,
ForwardIterator first2, ForwardIterator last2,
BinaryPredicate pred);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator1
find_first_of(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2,
    BinaryPredicate pred);

Effects: Finds an element that matches one of a set of values.

Returns: The first iterator \(i\) in the range \([first1, last1)\) such that for some iterator \(j\) in the range \([first2, last2)\) the following conditions hold: \(*i == *j, pred(*i,*j) != false\). Returns \(last1\) if \([first2, last2)\) is empty or if no such iterator is found.

Complexity: At most \((last1-first1) \times (last2-first2)\) applications of the corresponding predicate.

28.5.8  Adjacent find  [alg.adjacent.find]

template<class ForwardIterator>
constexpr ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
adjacent_find(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last);

Effects: Returns the first iterator \(i\) such that both \(i\) and \(i + 1\) are in the range \([first, last)\) for which the following corresponding conditions hold: \(*i == *(i + 1)\), \(pred(*i, *(i + 1)) != false\). Returns \(last\) if no such iterator is found.

Complexity: For the overloads with no \(ExecutionPolicy\), exactly \(\min((i - first) + 1, (last - first) - 1)\) applications of the corresponding predicate. For the overloads with an \(ExecutionPolicy\), \(\Theta(last - first)\) applications of the corresponding predicate.

28.5.9  Count  [alg.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 InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last, const T& value);

template<class InputIterator, class Predicate>
constexpr typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);

template<class ExecutionPolicy, class InputIterator, class Predicate>
typename iterator_traits<InputIterator>::difference_type
count_if(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last, Predicate pred);

Effects: Returns the number of iterators \(i\) in the range \([first, last)\) for which the following corresponding conditions hold: \(*i == value, pred(*i) != false\).

Complexity: Exactly \(last - first\) applications of the corresponding predicate.
28.5.10  Mismatch

\[\text{template<class InputIterator1, class InputIterator2>}
\]
\[\text{constexpr pair<InputIterator1, InputIterator2>}
\]
\[\text{mismatch(InputIterator1 first1, InputIterator1 last1,}
\]
\[\text{InputIterator2 first2);}\]
\[\text{template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>}
\]
\[\text{pair<ForwardIterator1, ForwardIterator2>}
\]
\[\text{mismatch(ExecutionPolicy&& exec,}
\]
\[\text{ForwardIterator1 first1, ForwardIterator1 last1,}
\]
\[\text{ForwardIterator2 first2);}\]
\[\text{template<class InputIterator1, class InputIterator2,}
\]
\[\text{class BinaryPredicate>}
\]
\[\text{constexpr pair<InputIterator1, InputIterator2>}
\]
\[\text{mismatch(InputIterator1 first1, InputIterator1 last1,}
\]
\[\text{InputIterator2 first2, BinaryPredicate pred);}\]
\[\text{template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,}
\]
\[\text{class BinaryPredicate>}
\]
\[\text{pair<ForwardIterator1, ForwardIterator2>}
\]
\[\text{mismatch(ExecutionPolicy&& exec,}
\]
\[\text{ForwardIterator1 first1, ForwardIterator1 last1,}
\]
\[\text{ForwardIterator2 first2, BinaryPredicate pred);}\]
\[\text{template<class InputIterator1, class InputIterator2,}
\]
\[\text{class BinaryPredicate>}
\]
\[\text{constexpr pair<InputIterator1, InputIterator2>}
\]
\[\text{mismatch(InputIterator1 first1, InputIterator1 last1,}
\]
\[\text{InputIterator2 first2, InputIterator2 last2,}
\]
\[\text{BinaryPredicate pred);}\]
\[\text{template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,}
\]
\[\text{class BinaryPredicate>}
\]
\[\text{pair<ForwardIterator1, ForwardIterator2>}
\]
\[\text{mismatch(ExecutionPolicy&& exec,}
\]
\[\text{ForwardIterator1 first1, ForwardIterator1 last1,}
\]
\[\text{ForwardIterator2 first2, ForwardIterator2 last2,}
\]
\[\text{BinaryPredicate pred);}\]

1  Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

2  Returns: A pair of iterators first1 + n and first2 + n, where n is the smallest integer such that, respectively,

\[\neg(*(first1 + n) == *(first2 + n)) \text{ or} \]
\[\neg(pred(*(first1 + n), *(first2 + n)) == false,\]

or \(\min(last1 - first1, last2 - first2)\) if no such integer exists.

3  Complexity: At most \(\min(last1 - first1, last2 - first2)\) applications of the corresponding predicate.

28.5.11  Equal

\[\text{template<class InputIterator1, class InputIterator2>}
\]
\[\text{constexpr bool equal(InputIterator1 first1, InputIterator1 last1,}
\]
\[\text{InputIterator2 first2);}\]

\[\text{§ 28.5.11} \quad 909\]
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, InputIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
bool equal(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, ForwardIterator2 last2);

template<class InputIterator1, class InputIterator2,
    class BinaryPredicate>
constexpr bool equal(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
bool equal(ExecutionPolicy&& exec,
    ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);

1 Remarks: If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

2 Returns: If last1 - first1 != last2 - first2, return false. Otherwise return true if for every iterator i in the range [first1, last1) the following corresponding conditions hold: *i == *(first2 + (i - first1)), pred(*i, *(first2 + (i - first1))) != false. Otherwise, returns false.

3 Complexity:
   (3.1) — For the overloads with no ExecutionPolicy,
       (3.1.1) — if InputIterator1 and InputIterator2 meet the requirements of random access iterators (27.2.7) and last1 - first1 != last2 - first2, then no applications of the corresponding predicate; otherwise,
       (3.1.2) — at most min(last1 - first1, last2 - first2) applications of the corresponding predicate.
   (3.2) — For the overloads with an ExecutionPolicy,
       (3.2.1) — if ForwardIterator1 and ForwardIterator2 meet the requirements of random access iterators and last1 - first1 != last2 - first2, then no applications of the corresponding predicate; otherwise,
       (3.2.2) — O(min(last1 - first1, last2 - first2)) applications of the corresponding predicate.

28.5.12 Is permutation [alg.is_permutation]

template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2);

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
    ForwardIterator2 first2, BinaryPredicate pred);
template<class ForwardIterator1, class ForwardIterator2>
constexpr bool is_permutation(ForwardIterator1 first1, ForwardIterator1 last1,
                                ForwardIterator2 first2, ForwardIterator2 last2);

1 \textit{Requires:} ForwardIterator1 and ForwardIterator2 shall have the same value type. The comparison function shall be an equivalence relation.

2 \textit{Remarks:} If last2 was not given in the argument list, it denotes first2 + (last1 - first1) below.

3 \textit{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.

4 \textit{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.

28.5.13 Search

\begin{verbatim}
template<class ForwardIterator1, class ForwardIterator2>
constexpr ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                                   ForwardIterator2 first2, ForwardIterator2 last2);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ExecutionPolicy&& exec,
                        ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
constexpr ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                                   ForwardIterator2 first2, ForwardIterator2 last2,
                                   BinaryPredicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(ExecutionPolicy&& exec,
                        ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        BinaryPredicate pred);
\end{verbatim}

1 \textit{Effects:} Finds a subsequence of equal values in a sequence.

2 \textit{Returns:} The first iterator \(i\) in the range \([first1, last1 - (last2-first2))\) such that for every non-negative integer \(n\) less than last2 - first2 the following corresponding conditions hold: \(*(i + n) == *(first2 + n), \text{pred}(*(i + n), *(first2 + n)) != \text{false}\.\) Returns first1 if \([first2, last2)\) is empty, otherwise returns last1 if no such iterator is found.

3 \textit{Complexity:} At most \((last1 - first1) \ast (last2 - first2)\) applications of the corresponding predicate.

\begin{verbatim}
template<class ForwardIterator, class Size, class T>
constexpr ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                                   Size count, const T& value);
\end{verbatim}
# 28.6 Mutating sequence operations

## 28.6.1 Copy

**template<class InputIterator, class OutputIterator>**

```cpp
cconstexpr OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);
```

1. **Requires:** result shall not be in the range [first, last).
2. **Effects:** Copies elements in the range [first, last) into the range [result, result + (last - first)) starting from first and proceeding to last. For each non-negative integer n < (last - first), performs *(result + n) = *(first + n).
3. **Returns:** result + (last - first).
4. **Complexity:** Exactly last - first assignments.

**template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>**

```cpp
ForwardIterator2 copy(ExecutionPolicy&& policy,
                      ForwardIterator1 first, ForwardIterator1 last,
                      ForwardIterator2 result);
```

5. **Requires:** The ranges [first, last) and [result, result + (last - first)) shall not overlap.
6. **Effects:** Copies elements in the range [first, last) into the range [result, result + (last - first)). For each non-negative integer n < (last - first), performs *(result + n) = *(first + n).
7. **Returns:** result + (last - first).
8. **Complexity:** Exactly last - 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);

**Effects:** For each non-negative integer \( i < n \), performs \( *(result + i) = *(first + i) \).

**Returns:** \( result + n \).

**Complexity:** Exactly \( n \) assignments.

template<class InputIterator, class OutputIterator, class Predicate>
constexpr OutputIterator copy_if(InputIterator first, InputIterator last,
OutputIterator result, Predicate pred);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class Predicate>
ForwardIterator2 copy_if(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, Predicate pred);

**Requires:** The ranges \([\text{first}, \text{last})\) and \([\text{result}, \text{result} + (\text{last} - \text{first}))\) shall not overlap.

**Note:** For the overload with an ExecutionPolicy, there may be a performance cost if `iterator_traits<ForwardIterator1>::value_type` is not `MoveConstructible` (Table 23). — end note

**Effects:** Copies all of the elements referred to by the iterator \( i \) in the range \([\text{first}, \text{last})\) for which \( \text{pred}(*i) \) is true.

**Returns:** The end of the resulting range.

**Complexity:** Exactly \( \text{last} - \text{first} \) applications of the corresponding predicate.

**Remarks:** Stable (20.5.5.7).

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
copy_backward(BidirectionalIterator1 first,
BidirectionalIterator1 last,
BidirectionalIterator2 result);

**Requires:** \( \text{result} \) shall not be in the range \([\text{first}, \text{last})\).

**Effects:** Copies elements in the range \([\text{first}, \text{last})\) into the range \([\text{result} - (\text{last} - \text{first}), \text{result})\) starting from \( \text{last} - 1 \) and proceeding to \( \text{first} \). For each positive integer \( n \leq (\text{last} - \text{first}) \), performs \( *(\text{result} - n) = *(\text{last} - n) \).

**Returns:** \( \text{result} - (\text{last} - \text{first}) \).

**Complexity:** Exactly \( \text{last} - \text{first} \) assignments.

### 28.6.2 Move

```cpp
template<class InputIterator, class OutputIterator>
constexpr OutputIterator move(InputIterator first, InputIterator last,
OutputIterator result);
```

**Requires:** \( \text{result} \) shall not be in the range \([\text{first}, \text{last})\).

**Effects:** Moves elements in the range \([\text{first}, \text{last})\) into the range \([\text{result}, \text{result} + (\text{last} - \text{first}))\) starting from first and proceeding to last. For each non-negative integer \( n < (\text{last} - \text{first}) \), performs \( *(\text{result} + n) = \text{std}::\text{move}(*(\text{first} + n)) \).

**Returns:** \( \text{result} + (\text{last} - \text{first}) \).

**Complexity:** Exactly \( \text{last} - \text{first} \) move assignments.

---

266) `copy_backward` should be used instead of `copy` when `last` is in the range \([\text{result} - (\text{last} - \text{first}), \text{result})\).

§ 28.6.2
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 move(ExecutionPolicy&& policy,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result);

Requires: The ranges [first, last) and [result, result + (last - first)) shall not overlap.

Effects: Moves elements in the range [first, last) into the range [result, result + (last - first)). For each non-negative integer n < (last - first), performs *(result + n) = std::move(*(first + n)).

Returns: result + (last - first).

Complexity: Exactly last - first assignments.

template<class BidirectionalIterator1, class BidirectionalIterator2>
constexpr BidirectionalIterator2
move_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
    BidirectionalIterator2 result);

Requires: result shall not be in the range (first, last).

Effects: Moves elements in the range [first, last) into the range [result - (last-first), result) starting from last - 1 and proceeding to first. For each positive integer n <= (last - first), performs *(result - n) = std::move(*(last - n)).

Returns: result - (last - first).

Complexity: Exactly last - first assignments.

28.6.3 Swap

template<class ForwardIterator1, class ForwardIterator2>
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);

Requires: The two ranges [first1, last1) and [first2, first2 + (last1 - first1)) shall not overlap. *(first1 + n) shall be swappable with (20.5.3.2) *(first2 + n).

Effects: For each non-negative integer n < (last1 - first1) performs: swap(*(first1 + n), *(first2 + n)).

Returns: first2 + (last1 - first1).

Complexity: Exactly last1 - first1 swaps.

template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);

Requires: a and b shall be dereferenceable. *a shall be swappable with (20.5.3.2) *b.

Effects: As if by swap(*a, *b).

28.6.4 Transform

template<class InputIterator, class OutputIterator,
    class UnaryOperation>
constexpr OutputIterator
transform(InputIterator first, InputIterator last,
    OutputIterator result, UnaryOperation op);

§ 28.6.4

(267) move_backward should be used instead of move when last is in the range [result - (last - first), result).
template<
class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class UnaryOperation>
ForwardIterator2
transform(ExecutionPolicy&& exec,
    ForwardIterator1 first, ForwardIterator1 last,
    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);

requires: op and binary_op shall not invalidate iterators or subranges, or modify elements in the
ranges
(1.1) — [first1, last1],
(1.2) — [first2, first2 + (last1 - first1)], and
(1.3) — [result, result + (last1 - first1)].

effects: Assigns through every iterator i in the range [result, result + (last1 - first1)) a
new corresponding value equal to op(*(first1 + (i - result))) or binary_op(*(first1 + (i -
result)), *(first2 + (i - result))).

returns: result + (last1 - first1).

complexity: Exactly last1 - first1 applications of op or binary_op. This requirement also applies
to the overload with an ExecutionPolicy.

remarks: result may be equal to first in case of unary transform, or to first1 or first2 in case of
binary transform.

28.6.5 Replace

[alg.replace]

template<class ForwardIterator, class T>
constexpr void replace(ForwardIterator first, ForwardIterator last,
    const T& old_value, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator, class T>
void replace(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    const T& old_value, const T& new_value);

template<class ForwardIterator, class Predicate, class T>
constexpr void replace_if(ForwardIterator first, ForwardIterator last,
    Predicate pred, const T& new_value);

template<class ExecutionPolicy, class ForwardIterator, class Predicate, class T>
void replace_if(ExecutionPolicy&& exec,
    ForwardIterator first, ForwardIterator last,
    Predicate pred, const T& new_value);

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 the following corresponding conditions hold: *i == old_value, pred(*i) != false.

complexity: Exactly last - first applications of the corresponding predicate.

(268) The use of fully closed ranges is intentional.
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);

4 Requires: The results of the expressions *first and new_value shall be writable (27.2.1) to the result output iterator. The ranges [first, last) and [result, result + (last - first)) shall not overlap.

5 Effects: Assigns to every iterator i in the range [result, result + (last - first)) either new_value or *(first + (i - result)) depending on whether the following corresponding conditions hold:
   *(first + (i - result)) == old_value
   pred(*(first + (i - result))) != false

6 Returns: result + (last - first).

7 Complexity: Exactly last - first applications of the corresponding predicate.

28.6.6 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);

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 Requires: The expression value shall be writable (27.2.1) to the output iterator. The type Size shall be convertible to an integral type (7.8, 15.3).

2 Effects: The fill algorithms assign value through all the iterators in the range [first, last). The fill_n algorithms assign value through all the iterators in the range [first, first + n) if n is positive, otherwise they do nothing.

3 Returns: fill_n returns first + n for non-negative values of n and first for negative values.

4 Complexity: Exactly last - first, n, or 0 assignments, respectively.

28.6.7 Generate

template<class ForwardIterator, class Generator>
constexpr void generate(ForwardIterator first, ForwardIterator last,
Generator gen);

    template<class ExecutionPolicy, class ForwardIterator, class Generator>
    void generate(ExecutionPolicy&& exec,
                  ForwardIterator first, ForwardIterator last,
                  Generator gen);

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);

    Requires: gen takes no arguments, Size shall be convertible to an integral type (7.8, 15.3).
    Effects: The generate algorithms invoke the function object gen and assign the return value of gen
             through all the iterators in the range [first, last). The generate_n algorithms invoke the function
             object gen and assign the return value of gen through all the iterators in the range [first, first +
             n) if n is positive, otherwise they do nothing.
    Returns: generate_n returns first + n for non-negative values of n and first for negative values.
    Complexity: Exactly last - first, n, or 0 invocations of gen and assignments, respectively.

28.6.8 Remove [alg.remove]

    template<class ForwardIterator, class T>
    constexpr ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                                      const T& value);

template<class ExecutionPolicy, class ForwardIterator, class T>
    ForwardIterator remove(ExecutionPolicy&& exec,
                            ForwardIterator first, ForwardIterator last,
                            const T& value);

template<class ForwardIterator, class Predicate>
    constexpr ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                                         Predicate pred);

template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    ForwardIterator remove_if(ExecutionPolicy&& exec,
                               ForwardIterator first, ForwardIterator last,
                               Predicate pred);

    Requires: The type of *first shall satisfy the MoveAssignable requirements (Table 25).
    Effects: Eliminates all the elements referred to by iterator i in the range [first, last) for which
             the following corresponding conditions hold: *i == value, pred(*i) != false.
    Returns: The end of the resulting range.
    Remarks: Stable (20.5.5.7).
    Complexity: Exactly last - first applications of the corresponding predicate.

    [Note: Each element in the range [ret, last), where ret is the returned value, has a valid but
        unspecified state, because the algorithms can eliminate elements by moving from elements that were
        originally in that range. — end note]

template<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);

Requires: The ranges [first, last) and [result, result + (last - first)) shall not overlap. 
The expression *result = *first shall be valid. [Note: For the overloads with an ExecutionPolicy, 
there may be a performance cost if iterator_traits<ForwardIterator1>::value_type is not Move- 
Constructible (Table 23). —end note]

Effects: Copies all the elements referred to by the iterator i in the range [first, last) for which the 
following corresponding conditions do not hold: *i == value, pred(*i) != false.

Returns: The end of the resulting range.

Complexity: Exactly last - first applications of the corresponding predicate.

Remarks: Stable (20.5.5.7).

28.6.9 Unique

§ 28.6.9

«alg.unique»
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryPredicate>
ForwardIterator2
unique_copy(ExecutionPolicy&& exec,
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result, BinaryPredicate pred);

Requires:
(5.1) The comparison function shall be an equivalence relation.
(5.2) The ranges [first, last) and [result, result+(last-first)) shall not overlap.
(5.3) The expression *result = *first shall be valid.
(5.4) For the overloads with no ExecutionPolicy, let T be the value type of InputIterator. If
InputIterator meets the forward iterator requirements, then there are no additional requirements
for T. Otherwise, if OutputIterator meets the forward iterator requirements and its value
type is the same as T, then T shall be CopyAssignable (Table 26). Otherwise, T shall be
both CopyConstructible (Table 24) and CopyAssignable. [Note: For the overloads with an
ExecutionPolicy, there may be a performance cost if the value type of ForwardIterator1 is
not both CopyConstructible and CopyAssignable. — end note]
rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);

template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator
rotate(ExecutionPolicy& exec,
       ForwardIterator first, ForwardIterator middle, ForwardIterator last);

1 Requires: [first, middle) and [middle, last) shall be valid ranges. ForwardIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 23) and the requirements of MoveAssignable (Table 25).

2 Effects: For each non-negative integer i < (last - first), places the element from the position first + i into position first + (i + (last - middle)) % (last - first).

3 Returns: first + (last - middle).

4 Remarks: This is a left rotate.

Complexity: At most last - first swaps.

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);

6 Requires: The ranges [first, last) and [result, result + (last - first)) shall not overlap.

7 Effects: Copies the range [first, last) to the range [result, result + (last - first)) such that for each non-negative integer i < (last - first) the following assignment takes place: *(result + i) = *(first + (i + (middle - first)) % (last - first)).

8 Returns: result + (last - first).

9 Complexity: Exactly last - first assignments.

28.6.12 Sample

template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomBitGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
                       SampleIterator out, Distance n,
                       UniformRandomBitGenerator& g);

1 Requires:

1.1 — PopulationIterator shall satisfy the requirements of an input iterator (27.2.3).
1.2 — SampleIterator shall satisfy the requirements of an output iterator (27.2.4).
1.3 — SampleIterator shall satisfy the additional requirements of a random access iterator (27.2.7) unless PopulationIterator satisfies the additional requirements of a forward iterator (27.2.5).
1.4 — PopulationIterator’s value type shall be writable (27.2.1) to out.
1.5 — Distance shall be an integer type.
1.6 — remove_reference_t<UniformRandomBitGenerator> shall meet the requirements of a uniform random bit generator type (29.6.1.3) whose return type is convertible to Distance.
1.7 — out shall not be in the range [first, last).

2 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 ]

3 Returns: The end of the resulting sample range.

Complexity: $\theta(last - first)$.

Remarks:
— Stable if and only if `PopulationIterator` satisfies the requirements of a forward iterator.

— 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.

### 28.6.13 Shuffle

```cpp
template<class RandomAccessIterator, class UniformRandomBitGenerator>
void shuffle(RandomAccessIterator first,
             RandomAccessIterator last,
             UniformRandomBitGenerator&& g);
```

1. **Requires:** `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (20.5.3.2). The type `remove_reference_t<UniformRandomBitGenerator>` shall meet the requirements of a uniform random bit generator (29.6.1.3) type whose return type is convertible to `iterator_traits<RandomAccessIterator>::difference_type`.

2. **Effects:** Permutes the elements in the range `[first, last)` such that each possible permutation of those elements has equal probability of appearance.

3. **Complexity:** Exactly `(last - first) - 1` swaps.

4. **Remarks:** 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.

### 28.7 Sorting and related operations

All the operations in 28.7 have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

`Compare` is a function object type (23.14). The return value of the function call operation applied to an object of type `Compare`, when contextually converted to `bool` (Clause 7), yields `true` if the first argument of the call is less than the second, and `false` otherwise. `Compare` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

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 28.7.3, `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:

1. `comp(a, b) && comp(b, c)` implies `comp(a, c)`
2. `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)`

[Note: Under these conditions, it can be shown that

3. `equiv` is an equivalence relation

4. `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

5. The induced relation is a strict total ordering.]

A sequence is *sorted with respect to a comparator `comp`* 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, `comp(*i + n, *i)` == `false`.

A sequence `[start, finish)` is *partitioned with respect to an expression `f(e)`* if there exists an integer `n` such that for all `0 <= i < (finish - start)`, `f(*((start + i)))` is `true` if and only if `i < n`.

In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.
28.7.1 Sorting

28.7.1.1 sort

```cpp
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator>
void sort(ExecutionPolicy&& exec,
          RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void sort(ExecutionPolicy&& exec,
          RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

1. **Requires:** RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

2. **Effects:** Sorts the elements in the range [first, last).

3. **Complexity:** \( \Theta(N \log N) \) comparisons, where \( N = \text{last} - \text{first} \).

28.7.1.2 stable_sort

```cpp
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator>
void stable_sort(ExecutionPolicy&& exec,
                 RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
void stable_sort(ExecutionPolicy&& exec,
                 RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

1. **Requires:** RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

2. **Effects:** Sorts the elements in the range [first, last).

3. **Complexity:** At most \( N \log^2(N) \) comparisons, where \( N = \text{last} - \text{first} \), but only \( N \log N \) comparisons if there is enough extra memory.

4. **Remarks:** Stable (20.5.5.7).

28.7.1.3 partial_sort

```cpp
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first,
                  RandomAccessIterator middle,
                  RandomAccessIterator last);

template<class ExecutionPolicy, class RandomAccessIterator>
void partial_sort(ExecutionPolicy&& exec,
                 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, class Compare>
void partial_sort(ExecutionPolicy&& exec,
                   RandomAccessIterator first,
                   RandomAccessIterator middle,
                   RandomAccessIterator last,
                   Compare comp);

1 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of *first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

2 Effects: Places the first middle - first sorted elements from the range [first, last) into the range [first, middle). The rest of the elements in the range [middle, last) are placed in an unspecified order.

3 Complexity: Approximately (last - first) * log(middle - first) comparisons.

28.7.1.4 partial_sort_copy

template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last);

template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator>
RandomAccessIterator
partial_sort_copy(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator,
         class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last,
                   Compare comp);

template<class ExecutionPolicy, class ForwardIterator, class RandomAccessIterator,
         class Compare>
RandomAccessIterator
partial_sort_copy(ExecutionPolicy&& exec,
                   ForwardIterator first, ForwardIterator last,
                   RandomAccessIterator result_first,
                   RandomAccessIterator result_last,
                   Compare comp);

1 Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of *result_first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

2 Effects: Places the first min(last - first, result_last - result_first) sorted elements into the range [result_first, result_first + min(last - first, result_last - result_first)).

3 Returns: The smaller of: result_last or result_first + (last - first).

4 Complexity: Approximately (last - first) * log(min(last - first, result_last - result_first)) comparisons.
28.7.1.5 is_sorted

template<class ForwardIterator>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last);

1 Returns: is_sorted_until(first, last) == last

template<class ExecutionPolicy, class ForwardIterator>
bool is_sorted(ExecutionPolicy&& exec,
              ForwardIterator first, ForwardIterator last);

2 Returns: is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last) == last

template<class ForwardIterator, class Compare>
constexpr bool is_sorted(ForwardIterator first, ForwardIterator last,
                         Compare comp);

3 Returns: is_sorted_until(first, last, comp) == last

template<class ExecutionPolicy, class ForwardIterator, class Compare>
bool is_sorted(ExecutionPolicy&& exec,
              ForwardIterator first, ForwardIterator last,
              Compare comp);

4 Returns: is_sorted_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == last

template<class ForwardIterator>
constexpr ForwardIterator
is_sorted_until(ForwardIterator first, ForwardIterator last);

5 Returns: If (last - first) < 2, returns last. Otherwise, returns the last iterator i in [first, last] for which the range [first, i) is sorted.

Complexity: Linear.

28.7.2 Nth element

template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

6 template<class ExecutionPolicy, class RandomAccessIterator>
void nth_element(ExecutionPolicy&& exec,
                 RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
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);

Requires: RandomAccessIterator shall satisfy the requirements of ValueSwappable \(^{(20.5.3.2)}\). The type of \(*\text{first}\) shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

Effects: After \(\text{nth_element}\) the element in the position pointed to by \(\text{nth}\) is the element that would be in that position if the whole range were sorted, unless \(\text{nth} == \text{last}\). Also for every iterator \(i\) in the range \([\text{first}, \text{nth})\) and every iterator \(j\) in the range \([\text{nth}, \text{last})\) it holds that: \(!(*j < *i)\) or \(\text{comp}(*j, *i) == \text{false}\).

Complexity: For the overloads with no \text{ExecutionPolicy}, linear on average. For the overloads with an \text{ExecutionPolicy}, \(\mathcal{O}(N)\) applications of the predicate, and \(\mathcal{O}(N \log N)\) swaps, where \(N = \text{last} - \text{first}\).  

28.7.3 Binary search \[^{[alg.binary.search]}\]

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 implied or explicit 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.

28.7.3.1 \text{lower_bound} \[^{[lower.bound]}\]

template<\text{class ForwardIterator, class T}> constexpr ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<\text{class ForwardIterator, class T, class Compare}> constexpr ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

Requires: The elements \(e\) of \([\text{first}, \text{last})\) shall be partitioned with respect to the expression \(e < \text{value}\) or \(\text{comp}(e, \text{value})\).

Returns: The furthermost iterator \(i\) in the range \([\text{first}, \text{last})\) such that for every iterator \(j\) in the range \([\text{first}, i)\) the following corresponding conditions hold: \(*j < \text{value}\) or \(\text{comp}(\*j, \text{value}) != \text{false}\).

Complexity: At most \(\log_2(\text{last} - \text{first}) + \mathcal{O}(1)\) comparisons.

28.7.3.2 \text{upper_bound} \[^{[upper.bound]}\]

template<\text{class ForwardIterator, class T}> constexpr ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last, const T& value);

template<\text{class ForwardIterator, class T, class Compare}> constexpr ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last, const T& value, Compare comp);

Requires: The elements \(e\) of \([\text{first}, \text{last})\) shall be partitioned with respect to the expression \(!(\text{value} < e)\) or \(!\text{comp}(\text{value}, e)\).

Returns: The furthermost iterator \(i\) in the range \([\text{first}, \text{last})\) such that for every iterator \(j\) in the range \([\text{first}, i)\) the following corresponding conditions hold: \(!\text{value} < *j\) or \(\text{comp}(\text{value}, *j) == \text{false}\).

Complexity: At most \(\log_2(\text{last} - \text{first}) + \mathcal{O}(1)\) comparisons.
28.7.3.3 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);

1 Requires: The elements e of [first, last) shall be partitioned with respect to the expressions e < value and !(value < e) or comp(e, value) and !comp(value, e). Also, for all elements e of [first, last), e < value shall imply !(value < e) or comp(e, value) shall imply !comp(value, e).

2 Returns:
make_pair(upper_bound(first, last, value),
          lower_bound(first, last, value))
or
make_pair(upper_bound(first, last, value, comp),
          lower_bound(first, last, value, comp))

3 Complexity: At most 2 * log₂(last - first) + Θ(1) comparisons.

28.7.3.4 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);

1 Requires: The elements e of [first, last) shall be partitioned with respect to the expressions e < value and !(value < e) or comp(e, value) and !comp(value, e). Also, for all elements e of [first, last), e < value shall imply !(value < e) or comp(e, value) shall imply !comp(value, e).

2 Returns: true if there is an iterator i in the range [first, last) that satisfies the corresponding conditions: !(i < value) && !comp(*i, value) == false && !comp(value, *i) == false.

3 Complexity: At most log₂(last - first) + Θ(1) comparisons.

28.7.4 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,
                ForwardIterator first, ForwardIterator last, Predicate pred);

1 Requires: For the overload with no ExecutionPolicy, InputIterator’s value type shall be convertible to Predicate’s argument type. For the overload with an ExecutionPolicy, ForwardIterator’s value type shall be convertible to Predicate’s argument type.

2 Returns: true if [first, last) is empty or if the elements e of [first, last) are partitioned with respect to the expression pred(e).

3 Complexity: Linear. At most last - first applications of pred.
template<class ForwardIterator, class Predicate>
    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);

    Requires: ForwardIterator shall satisfy the requirements of ValueSwappable (20.5.3.2).
    Effects: Places all the elements in the range [first, last) that satisfy pred before all the elements
    that do not satisfy it.
    Returns: An iterator i such that for every iterator j in the range [first, i) pred(*j) != false, and for every iterator k in the range [i, last), pred(*k) == false.
    Complexity: Let N = last - first:
    — For the overload with no ExecutionPolicy, exactly N applications of the predicate. At most
      N/2 swaps if ForwardIterator meets the BidirectionalIterator requirements and at most N
      swaps otherwise.
    — 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);

    Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The
    type of *first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable
    (Table 25).
    Effects: Places all the elements in the range [first, last) that satisfy pred before all the elements
    that do not satisfy it.
    Returns: An iterator i such that for every iterator j in the range [first, i), pred(*j) != false, and for every iterator k in the range [i, last), pred(*k) == false. The relative order of the elements
    in both groups is preserved.
    Complexity: Let N = last - first:
    — For the overload 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.
    — 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);

    Requires:
    — For the overload with no ExecutionPolicy, InputIterator's value type shall be CopyAssignable
      (Table 26), and shall be writable (27.2.1) to the out_true and out_false OutputIterators, and
      shall be convertible to Predicate's argument type.

§ 28.7.4
For the overload with an ExecutionPolicy, ForwardIterator’s value type shall be CopyAssignable, and shall be writable to the out_true and out_false ForwardIterators, and shall be convertible to Predicate’s argument type. [Note: There may be a performance cost if ForwardIterator's value type is not CopyConstructible. — end note]

For both overloads, the input range shall not overlap with either of the output ranges.

Effects: For each iterator i in [first, last), copies *i to the output range beginning with out_true if pred(*i) is true, or to the output range beginning with out_false otherwise.

Returns: A pair p such that p.first is the end of the output range beginning at out_true and p.second is the end of the output range beginning at out_false.

Complexity: Exactly last - first applications of pred.

template<class ForwardIterator, class Predicate>
constexpr ForwardIterator
partition_point(ForwardIterator first, ForwardIterator last, Predicate pred);

Requires: ForwardIterator’s value type shall be convertible to Predicate’s argument type. The elements e of [first, last) shall be partitioned with respect to the expression pred(e).

Returns: An iterator mid such that all_of(first, mid, pred) and none_of(mid, last, pred) are both true.

Complexity: \( O(\log(last - first)) \) applications of pred.

28.7.5 Merge [alg.merge]

template<class InputIterator1, class InputIterator2, class OutputIterator>
constexpr OutputIterator
merge(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2, OutputIterator result);

Requires: The ranges [first1, last1) and [first2, last2) shall be sorted with respect to operator< or comp. 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 is result + (last1 - first1) + (last2 - first2), such that the resulting range satisfies is_sorted(result, result_last) or is_sorted(result, result_last, comp), respectively.

Returns: result + (last1 - first1) + (last2 - first2).

Complexity: Let \( N = (last1 - first1) + (last2 - first2) \):

(4.1) — For the overloads with no ExecutionPolicy, at most \( N - 1 \) comparisons.
For the overloads with an `ExecutionPolicy`, \( O(N) \) comparisons.

Remarks: Stable (20.5.5.7).

```cpp
#template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last);
#template<class ExecutionPolicy, class BidirectionalIterator>
void inplace_merge(ExecutionPolicy&& exec,
                  BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last);
#template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last, Compare comp);
#template<class ExecutionPolicy, class BidirectionalIterator, class Compare>
void inplace_merge(ExecutionPolicy&& exec,
                  BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last, Compare comp);
```

Requires: The ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\) shall be sorted with respect to `operator<` or `comp`. `BidirectionalIterator` shall satisfy the requirements of `ValueSwappable` (20.5.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 23) and of `MoveAssignable` (Table 25).

Effects: Merges two sorted consecutive ranges \([\text{first}, \text{middle})\) and \([\text{middle}, \text{last})\), putting the result of the merge into the range \([\text{first}, \text{last})\). The resulting range will be in non-decreasing order; that is, for every iterator \(i\) in \([\text{first}, \text{last})\) other than `first`, the condition `*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))` will be false.

Complexity: Let \(N = \text{last} - \text{first}:

(8.1) — For the overloads with no `ExecutionPolicy`, if enough additional memory is available, exactly \(N - 1\) comparisons.

(8.2) — For the overloads with no `ExecutionPolicy` if no additional memory is available, \(O(N \log N)\) comparisons.

(8.3) — For the overloads with an `ExecutionPolicy`, \(O(N \log N)\) comparisons.

Remarks: Stable (20.5.5.7).

28.7.6 Set operations on sorted structures

This subclause defines all the basic set operations on sorted structures. They also work with `multisets` (26.4.7) 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.

28.7.6.1 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 ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class Compare>
bool includes(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
ForwardIterator2 first2, ForwardIterator2 last2,
Compare comp);

1 Returns: true if [first2, last2) is empty or if every element in the range [first2, last2) is contained in the range [first1, last1). Returns false otherwise.
2 Complexity: At most $2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ comparisons.

28.7.6.2 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);

1 Requires: The resulting range shall not overlap with either of the original ranges.
2 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.
3 Returns: The end of the constructed range.
4 Complexity: At most $2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1$ comparisons.
5 Remarks: If [first1, last1) contains $m$ elements that are equivalent to each other and [first2, last2) contains $n$ elements that are equivalent to them, then all $m$ elements from the first range shall be copied to the output range, in order, and then max($n - m, 0$) elements from the second range shall be copied to the output range, in order.

28.7.6.3 set_intersection

template<class InputIterator1, class InputIterator2,
class OutputIterator>
constexpr OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2,
OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
class ForwardIterator>
ForwardIterator
set_intersection(ExecutionPolicy&& exec,
ForwardIterator1 first1, ForwardIterator1 last1,
template<class InputIterator1, class InputIterator2, 
               class OutputIterator, class Compare>
  constexpr OutputIterator
  set_intersection(InputIterator1 first1, InputIterator1 last1, 
                   InputIterator2 first2, InputIterator2 last2, 
                   OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
         class ForwardIterator, class Compare>
  ForwardIterator
  set_intersection(ExecutionPolicy&& exec, 
                   ForwardIterator1 first1, ForwardIterator1 last1, 
                   ForwardIterator2 first2, ForwardIterator2 last2, 
                   ForwardIterator result, Compare comp);

Requires: The resulting range shall not overlap with either of the original ranges.

Effects: Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements 
that are present in both of the ranges.

Returns: The end of the constructed range.

Complexity: At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) comparisons.

Remarks: If \([first1, last1)\) contains \(m\) elements that are equivalent to each other and \([first2, \) \(last2)\) contains \(n\) elements that are equivalent to them, the first \(\min(m, n)\) elements shall be copied 
from the first range to the output range, in order.

28.7.6.4 set_difference

template<class InputIterator1, class InputIterator2, 
         class OutputIterator>
  constexpr OutputIterator
  set_difference(InputIterator1 first1, InputIterator1 last1, 
                 InputIterator2 first2, InputIterator2 last2, 
                 OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
          class ForwardIterator>
  ForwardIterator
  set_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_difference(InputIterator1 first1, InputIterator1 last1, 
                 InputIterator2 first2, InputIterator2 last2, 
                 OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, 
          class ForwardIterator, class Compare>
  ForwardIterator
  set_difference(ExecutionPolicy&& exec, 
                 ForwardIterator1 first1, ForwardIterator1 last1, 
                 ForwardIterator2 first2, ForwardIterator2 last2, 
                 ForwardIterator result, Compare comp);

Requires: The resulting range shall not overlap with either of the original ranges.

Effects: Copies the elements of the range \([first1, last1)\) which are not present in the range 
\([first2, last2)\) to the range beginning at \(\text{result}\). The elements in the constructed range are sorted.

Returns: The end of the constructed range.

Complexity: At most \(2 \times ((last1 - first1) + (last2 - first2)) - 1\) comparisons.
5 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 \([\text{first1}, \text{last1})\) shall be copied to the output range.

28.7.6.5 \texttt{set\_symmetric\_difference} \[set.symmetric.difference\]

\begin{verbatim}
template<class InputIterator1, class InputIterator2,
        class OutputIterator>
constexpr OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec,
                        ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        ForwardIterator result);

template<class InputIterator1, class InputIterator2,
        class OutputIterator, class Compare>
constexpr OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class ForwardIterator, class Compare>
ForwardIterator
set_symmetric_difference(ExecutionPolicy&& exec,
                        ForwardIterator1 first1, ForwardIterator1 last1,
                        ForwardIterator2 first2, ForwardIterator2 last2,
                        ForwardIterator result, Compare comp);
\end{verbatim}

1 Requires: The resulting range shall not overlap with either of the original ranges.

2 Effects: Copies the elements of the range \([\text{first1}, \text{last1})\) that are not present in the range \([\text{first2}, \text{last2})\), and the elements of the range \([\text{first2}, \text{last2})\) that are not present in the range \([\text{first1}, \text{last1})\) to the range beginning at \(\text{result}\). The elements in the constructed range are sorted.

3 Returns: The end of the constructed range.

4 Complexity: At most \(2 \times ((\text{last1} - \text{first1}) + (\text{last2} - \text{first2})) - 1\) comparisons.

5 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, then \(|m - n|\) of those elements shall be copied to the output range: the last \(m - n\) of these elements from \([\text{first1}, \text{last1})\) if \(m > n\), and the last \(n - m\) of these elements from \([\text{first2}, \text{last2})\) if \(m < n\).

28.7.7 \textbf{Heap operations} \[alg.heap.operations\]

1 A heap is a particular organization of elements in a range between two random access iterators \([\text{a}, \text{b})\) such that:

\begin{enumerate}
\item With \(N = \text{b} - \text{a}\), for all \(i, 0 < i < N\), \(\text{comp}(\text{a}[\lfloor \frac{i - 1}{2} \rfloor]), \text{a}[i])\) is \texttt{false}.
\item \texttt{a} may be removed by \texttt{pop_heap()}, or a new element added by \texttt{push_heap()}, in \(O(\log N)\) time.
\end{enumerate}

2 These properties make heaps useful as priority queues.

3 \texttt{make_heap()} converts a range into a heap and \texttt{sort_heap()} turns a heap into a sorted sequence.

28.7.7.1 \texttt{push\_heap} \[push.heap\]

\begin{verbatim}
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
\end{verbatim}
template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

Requires: The range \([\text{first}, \text{last} - 1]\) shall be a valid heap. The type of \*first shall satisfy the MoveConstructible requirements (Table 23) and the MoveAssignable requirements (Table 25).

Effects: Places the value in the location last - 1 into the resulting heap [first, last).

Complexity: At most \log(last - first) comparisons.

28.7.7.2 pop_heap

template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

Requires: The range \([\text{first}, \text{last})\) shall be a valid non-empty heap. RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of \*first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

Effects: Swaps the value in the location first with the value in the location last - 1 and makes [first, last - 1) into a heap.

Complexity: At most 2\log(last - first) comparisons.

28.7.7.3 make_heap

template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

Requires: The type of \*first shall satisfy the MoveConstructible requirements (Table 23) and the MoveAssignable requirements (Table 25).

Effects: Constructs a heap out of the range [first, last).

Complexity: At most 3(last - first) comparisons.

28.7.7.4 sort_heap

template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
    Compare comp);

Requires: The range \([\text{first}, \text{last})\) shall be a valid heap. RandomAccessIterator shall satisfy the requirements of ValueSwappable (20.5.3.2). The type of \*first shall satisfy the requirements of MoveConstructible (Table 23) and of MoveAssignable (Table 25).

Effects: Sorts elements in the heap [first, last).

Complexity: At most \(2N \log N\) comparisons, where \(N = \text{last} - \text{first}\).

28.7.7.5 is_heap

template<class RandomAccessIterator>
constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last);

Returns: is_heap_until(first, last) == last

\[\text{§ 28.7.7.5}\]
template<class ExecutionPolicy, class RandomAccessIterator>
bool is_heap(ExecutionPolicy&& exec,
            RandomAccessIterator first, RandomAccessIterator last);

Returns: is_heap_until(std::forward<ExecutionPolicy>(exec), first, last) == last

template<class RandomAccessIterator, class Compare>
constexpr bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
                        Compare comp);

Returns: is_heap_until(first, last, comp) == last

template<class ExecutionPolicy, class RandomAccessIterator, class Compare>
bool is_heap(ExecutionPolicy&& exec,
             RandomAccessIterator first, RandomAccessIterator last,
             Compare comp);

Returns:
    is_heap_until(std::forward<ExecutionPolicy>(exec), first, last, comp) == 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);

Returns: If (last - first) < 2, returns last. Otherwise, returns the last iterator i in
[first, last] for which the range [first, i) is a heap.

Complexity: Linear.

28.7.8 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);

Requires: For the first form, type T shall be LessThanComparable (Table 21).

Returns: The smaller value.

Remarks: Returns the first argument when the arguments are equivalent.

Complexity: Exactly one comparison.

template<class T>
constexpr T min(initializer_list<T> t);

template<class T, class Compare>
constexpr T min(initializer_list<T> t, Compare comp);

Requires: T shall be CopyConstructible and t.size() > 0. For the first form, type T shall be
LessThanComparable.

Returns: The smallest value in the initializer_list.

Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the
smallest.

Complexity: Exactly t.size() - 1 comparisons.
template<class T> constexpr const T& max(const T& a, const T& b);

```cpp
template<class T, class Compare>
constexpr const T& max(const T& a, const T& b, Compare comp);
```

Requires: For the first form, type T shall be LessThanComparable (Table 21).
Returns: The larger value.
Remarks: Returns the first argument when the arguments are equivalent.
Complexity: Exactly one comparison.

```cpp
template<class T>
constexpr T max(initializer_list<T> t);
```

```cpp
template<class T, class Compare>
constexpr T max(initializer_list<T> t, Compare comp);
```

Requires: T shall be CopyConstructible and t.size() > 0. For the first form, type T shall be LessThanComparable.
Returns: The largest value in the initializer_list.
Remarks: Returns a copy of the leftmost argument when several arguments are equivalent to the largest.
Complexity: Exactly t.size() - 1 comparisons.

template<class T> constexpr pair<const T&, const T&> minmax(const T& a, const T& b);

```cpp
template<class T, class Compare>
constexpr pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

Requires: For the first form, type T shall be LessThanComparable (Table 21).
Returns: pair<const T&, const T&>(b, a) if b is smaller than a, and pair<const T&, const T&>(a, b) otherwise.
Remarks: Returns pair<const T&, const T&>(a, b) when the arguments are equivalent.
Complexity: Exactly one comparison.

```cpp
template<class T>
constexpr pair<T, T> minmax(initializer_list<T> t);
```

```cpp
template<class T, class Compare>
constexpr pair<T, T> minmax(initializer_list<T> t, Compare comp);
```

Requires: T shall be CopyConstructible and t.size() > 0. For the first form, type T shall be LessThanComparable.
Returns: pair<T, T>(x, y), where x has the smallest and y has the largest value in the initializer list.
Remarks: x is a copy of the leftmost argument when several arguments are equivalent to the smallest.
y is a copy of the rightmost argument when several arguments are equivalent to the largest.
Complexity: At most (3/2)t.size() applications of the corresponding predicate.

```cpp
template<class ForwardIterator>
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator>
ForwardIterator min_element(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last);
```

```cpp
template<class ForwardIterator, class Compare>
constexpr ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

```cpp
template<class ExecutionPolicy, class ForwardIterator, class Compare>
ForwardIterator min_element(ExecutionPolicy&& exec,
ForwardIterator first, ForwardIterator last,
Compare comp);
```

Returns: The first iterator i in the range [first, last) such that for every iterator j in the range [first, last) the following corresponding conditions hold: !(j < *i) or comp(*j, *i) == false. Returns last if first == last.

§ 28.7.8
26 Complexity: Exactly \(\text{max}(\text{last} - \text{first} - 1, 0)\) applications of the corresponding comparisons.

\[
\text{template\text{<class ForwardIterator>}}
\text{constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last);} \\
\text{template\text{<class ExecutionPolicy, class ForwardIterator>}}
\text{ForwardIterator max_element(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last);} \\
\text{template\text{<class ForwardIterator, class Compare>}}
\text{constexpr ForwardIterator max_element(ForwardIterator first, ForwardIterator last,} \\
\text{Compare comp);} \\
\text{template\text{<class ExecutionPolicy, class ForwardIterator, class Compare>}}
\text{ForwardIterator max_element(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last,} \\
\text{Compare comp);} \\
\]

27 Returns: The first iterator \(i\) in the range \([\text{first}, \text{last})\) such that for every iterator \(j\) in the range \([\text{first}, \text{last})\) the following corresponding conditions hold: \(!(*i < *j)\) or \(\text{comp(*i, *j)} == \text{false}\). Returns \(\text{last}\) if \(\text{first} == \text{last}\).

28 Complexity: Exactly \(\text{max}(\text{last} - \text{first} - 1, 0)\) applications of the corresponding comparisons.

\[
\text{template\text{<class ForwardIterator>}}
\text{constexpr pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last);} \\
\text{template\text{<class ExecutionPolicy, class ForwardIterator>}}
\text{pair<ForwardIterator, ForwardIterator> minmax_element(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last);} \\
\text{template\text{<class ForwardIterator, class Compare>}}
\text{constexpr pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first, ForwardIterator last,} \\
\text{Compare comp);} \\
\text{template\text{<class ExecutionPolicy, class ForwardIterator, class Compare>}}
\text{pair<ForwardIterator, ForwardIterator> minmax_element(ExecutionPolicy&& exec,} \\
\text{ForwardIterator first, ForwardIterator last,} \\
\text{Compare comp);} \\
\]

29 Returns: \(\text{make\_pair}(\text{first}, \text{first})\) if \([\text{first}, \text{last})\) is empty, otherwise \(\text{make\_pair}(m, M)\), where \(m\) is the first iterator in \([\text{first}, \text{last})\) such that no iterator in the range refers to a smaller element, and where \(M\) is the last iterator\(^{269}\) in \([\text{first}, \text{last})\) such that no iterator in the range refers to a larger element.

30 Complexity: At most \(\text{max}(\lfloor \frac{3}{2}(N - 1) \rfloor, 0)\) applications of the corresponding predicate, where \(N\) is \(\text{last} - \text{first}\).

28.7.9 Bounded value

\[
\text{template\text{<class T>}}
\text{constexpr const T& clamp(const T& v, const T& lo, const T& hi);} \\
\text{template\text{<class T, class Compare>}}
\text{constexpr const T& clamp(const T& v, const T& lo, const T& hi, Compare comp);} \\
\]

1 Requires: The value of \(\text{lo}\) shall be no greater than \(\text{hi}\). For the first form, type \(T\) shall be \text{LessThanComparable} (Table 21).

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.

28.7.10 Lexicographical comparison

\[
\text{template\text{<class InputIterator1, class InputIterator2>}}
\text{constexpr bool} \\
\]

269) This behavior intentionally differs from \text{max\_element}().
```cpp
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 the sequence of elements defined by the range [first1, last1) is lexicographically less than the sequence of elements defined by the range [first2, last2) and false otherwise.

2 Complexity: At most $2 \min(last1 - first1, last2 - first2)$ applications of the corresponding comparison.

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 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.

4 [Example: The following sample implementation satisfies these requirements:
```cpp
for ( ; first1 != last1 && first2 != last2 ; ++first1, (void) ++first2) {
    if (*first1 < *first2) return true;
    if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
— end example]
```

5 [Note: An empty sequence is lexicographically less than any non-empty sequence, but not less than any empty sequence. — end note]

### 28.7.11 Three-way comparison algorithms

```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 template<class InputIterator1, class InputIterator2>
  constexpr auto
  lexicographical_compare_3way(InputIterator1 b1, InputIterator1 e1,
                               InputIterator2 b2, InputIterator2 e2);

4 Effects: Equivalent to:

```
return lexicographical_compare_3way(b1, e1, b2, e2,
[](const auto& t, const auto& u) { return compare_3way(t, u); });
```

28.7.12 Permutation generators

```cpp
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);
```

1 Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (20.5.3.2).

2 Effects: Takes a sequence defined by the range [first, last) and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to operator< or comp.

3 Returns: true if such a permutation exists. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns false.

4 Complexity: At most (last - first) / 2 swaps.

```cpp
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);
```

5 Requires: BidirectionalIterator shall satisfy the requirements of ValueSwappable (20.5.3.2).

6 Effects: Takes a sequence defined by the range [first, last) and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to operator< or comp.

7 Returns: true if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns false.

8 Complexity: At most (last - first) / 2 swaps.

28.8 C library algorithms

```cpp
void* bsearch(const void* key, const void* base, size_t nmemb, size_t size,
               c-compare-pred* compar);
```

[Note: The header <cstdlib> (21.2.2) declares the functions described in this subclause. — end note]
Effects: These functions have the semantics specified in the C standard library.

Remarks: The behavior is undefined unless the objects in the array pointed to by base are of trivial type.

Throws: Any exception thrown by compar() (20.5.5.12).

See also: ISO C 7.22.5.
29 Numerics library

29.1 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 101.

Table 101 — Numerics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.2 Definitions</td>
<td></td>
</tr>
<tr>
<td>29.3 Requirements</td>
<td></td>
</tr>
<tr>
<td>29.4 Floating-point environment</td>
<td>&lt;cfenv&gt;</td>
</tr>
<tr>
<td>29.5 Complex numbers</td>
<td>&lt;complex&gt;</td>
</tr>
<tr>
<td>29.6 Random number generation</td>
<td>&lt;random&gt;</td>
</tr>
<tr>
<td>29.7 Numeric arrays</td>
<td>&lt;valarray&gt;</td>
</tr>
<tr>
<td>29.8 Generalized numeric operations</td>
<td>&lt;numeric&gt;</td>
</tr>
<tr>
<td>29.9 Mathematical functions for floating-point types</td>
<td>&lt;cmath&gt;, &lt;cstdlib&gt;</td>
</tr>
</tbody>
</table>

29.2 Definitions

Define \( \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, a_1, \ldots, a_N) \) as follows:

1. \( a_1 \) when \( N \) is 1, otherwise
2. \( \text{op}(\text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, a_1, \ldots, a_K), \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, a_M, \ldots, a_N)) \) for any \( K \) where \( 1 < K + 1 = M \leq N \).

Define \( \text{GENERALIZED\_SUM}(\text{op}, a_1, \ldots, a_N) \) as \( \text{GENERALIZED\_NONCOMMUTATIVE\_SUM}(\text{op}, b_1, \ldots, b_N) \), where \( b_1, \ldots, b_N \) may be any permutation of \( a_1, \ldots, a_N \).

29.3 Numeric type requirements

The \text{complex} and \text{valarray} components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a type \( T \) that satisfies the following requirements:

1. \( T \) is not an abstract class (it has no pure virtual member functions);
2. \( T \) is not a reference type;
3. \( T \) is not cv-qualified;
4. If \( T \) is a class, it has a public default constructor;
5. If \( T \) is a class, it has a public copy constructor with the signature \( T::T(const T&) \);
6. If \( T \) is a class, it has a public destructor;
7. If \( T \) is a class, it has a public assignment operator whose signature is either \( T& T::operator=(const T&) \) or \( T& T::operator=(T) \);
8. If \( T \) is a class, its assignment operator, copy and default constructors, and destructor shall correspond to each other in the following sense:

\[ \text{Initialization of raw storage using the copy constructor on the value of } T(), \text{ however obtained, is semantically equivalent to value-initialization of the same raw storage.} \]

In other words, value types. These include arithmetic types, pointers, the library class \text{complex}, and instantiations of \text{valarray} for value types.

\[ \text{§ 29.3} \]
Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor.

Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.

[Note: This rule states, in part, that there shall not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized.]

[Example: An implementation is allowed to initialize a valarray by allocating storage using the new operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. — end example]

If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use vector (26.3.11) instead of valarray for that class. — end note]

If T is a class, it does not overload unary operator&.

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]

29.4 The floating-point environment [cfenv]

29.4.1 Header <cfenv> synopsis [cfenv.syn]

```c++
#define FE_ALL_EXCEPT see below
#define FE_DIVBYZERO see below
#define FE_INEXACT see below
#define FE_INVALID see below
#define FE_OVERFLOW see below
#define FE_UNDERFLOW see below

#define FE_DOWNWARD see below
#define FE_TONEAREST see below
#define FE_TOWARDZERO see below
#define FE_UPWARD see below
#define FE_DFL_ENV see below

namespace std {

// types
using fenv_t = object type;
using fexcept_t = integer type;

// functions
int feclearexcept(int except);
int fegetexceptflag(fexcept_t* pflag, int except);
int feraiseexcept(int except);
int fesetexceptflag(const fexcept_t* pflag, int except);
int fetestexcept(int except);
int fegetround();
int fesetround(int mode);
int fegetenv(fenv_t* penv);
int feholdexcept(fenv_t* penv);
int fetenv(const fenv_t* penv);
int feupdateenv(const fenv_t* penv);
}
```

§ 29.4.1
The contents and meaning of the header `<cfenv>` are the same as the C standard library header `<fenv.h>.

[Note: This document does not require an implementation to support the FENV_ACCESS pragma; it is implementation-defined (19.6) 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 (6.6.4.2). 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 (33.3.2) 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

29.5 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 expression 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]`.

29.5.1 Header `<complex>` synopsis

```cpp
namespace std {
    // 29.5.2, class template complex
    template<class T> class complex;

    // 29.5.3, complex specializations
    template<> class complex<float>;
    template<> class complex<double>;
    template<> class complex<long double>;

    // 29.5.6, operators
    template<class T> constexpr complex<T> operator+(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator+(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator+(const T&, const complex<T>&);

    template<class T> constexpr complex<T> operator-(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator-(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator-(const T&, const complex<T>&);

    template<class T> constexpr complex<T> operator*(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator*(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator*(const T&, const complex<T>&);

    template<class T> constexpr complex<T> operator/(const complex<T>&, const complex<T>&);
    template<class T> constexpr complex<T> operator/(const complex<T>&, const T&);
    template<class T> constexpr complex<T> operator/(const T&, const complex<T>&);
}
```
template<class T> constexpr complex<T> operator/(const complex<T>&, const complex<T>&);

template<class T> constexpr complex<T> operator/(const complex<T>&, const T&);

template<class T> constexpr complex<T> operator/(const T&, const complex<T>&);

template<class T> constexpr complex<T> operator+(const complex<T>&);

template<class T> constexpr complex<T> operator-(const complex<T>&);

template<class T> constexpr bool operator==(const complex<T>&, const complex<T>&);

template<class T> constexpr bool operator==(const complex<T>&, const T&);

template<class T> constexpr bool operator==(const T&, const complex<T>&);

template<class T> constexpr bool operator!=(const complex<T>&, const complex<T>&);

template<class T> constexpr bool operator!=(const complex<T>&, const T&);

template<class T> constexpr bool operator!=(const T&, const complex<T>&);

template<class T, class charT, class traits>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&, complex<T>&);

template<class T, class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&, const complex<T>&);

// 29.5.7, values
template<class T> constexpr T real(const complex<T>&);

template<class T> constexpr T imag(const complex<T>&);

template<class T> T abs(const complex<T>&);

template<class T> T arg(const complex<T>&);

template<class T> constexpr T norm(const complex<T>&);

template<class T> constexpr complex<T> conj(const complex<T>&);

template<class T> complex<T> proj(const complex<T>&);

template<class T> complex<T> polar(const T&, const T& = T());

// 29.5.8, transcendentals
template<class T> complex<T> acos(const complex<T>&);

template<class T> complex<T> asin(const complex<T>&);

template<class T> complex<T> atan(const complex<T>&);

template<class T> complex<T> acosh(const complex<T>&);

template<class T> complex<T> asinh(const complex<T>&);

template<class T> complex<T> atanh(const complex<T>&);

template<class T> complex<T> cos (const complex<T>&);

template<class T> complex<T> cosh (const complex<T>&);

template<class T> complex<T> exp (const complex<T>&);

template<class T> complex<T> log (const complex<T>&);

template<class T> complex<T> log10(const complex<T>&);

template<class T> complex<T> pow (const complex<T>&, const T&);

template<class T> complex<T> pow (const complex<T>&, const complex<T>&);

template<class T> complex<T> pow (const T&, const complex<T>&);

template<class T> complex<T> sin (const complex<T>&);

template<class T> complex<T> sinh (const complex<T>&);

template<class T> complex<T> sqrt (const complex<T>&);

template<class T> complex<T> tan (const complex<T>&);

template<class T> complex<T> tanh (const complex<T>&);

// 29.5.10, 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);
   }
The class complex describes an object that can store the Cartesian components, \( \text{real}() \) and \( \text{imag}() \), of a complex number.

### 29.5.3 complex specializations

```cpp
namespace std {
    template<> class complex<float> {
        public:
            using value_type = float;

            constexpr complex(float re = 0.0f, float im = 0.0f);
            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);

        };
    }
}```
template<class X> constexpr complex& operator-=(const complex<X>&);
template<class X> constexpr complex& operator*=(const complex<X>&);
template<class X> constexpr complex& operator/=(const complex<X>&);

};

template<> class complex<double> {
public:
    using value_type = double;

    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(const complex<float>&);
    constexpr explicit complex(const complex<long double>&);

    constexpr double real() const;
    constexpr void real(double);
    constexpr double imag() const;
    constexpr void imag(double);

    constexpr complex& operator= (double);
    constexpr complex& operator+=(double);
    constexpr complex& operator-=(double);
    constexpr complex& operator*=(double);
    constexpr complex& operator/=(double);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator= (const complex<X>&);
    template<class X> constexpr complex& operator+=(const complex<X>&);
    template<class X> constexpr complex& operator-=(const complex<X>&);
    template<class X> constexpr complex& operator*=(const complex<X>&);
    template<class X> constexpr complex& operator/=(const complex<X>&);

};

template<> class complex<long double> {
public:
    using value_type = long double;

    constexpr complex(long double re = 0.0L, long double im = 0.0L);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&);

    constexpr long double real() const;
    constexpr void real(long double);
    constexpr long double imag() const;
    constexpr void imag(long double);

    constexpr complex& operator= (long double);
    constexpr complex& operator+=(long double);
    constexpr complex& operator-=(long double);
    constexpr complex& operator*=(long double);
    constexpr complex& operator/=(long double);

    constexpr complex& operator=(const complex&);
    template<class X> constexpr complex& operator= (const complex<X>&);
    template<class X> constexpr complex& operator+=(const complex<X>&);
    template<class X> constexpr complex& operator-=(const complex<X>&);
    template<class X> constexpr complex& operator*=(const complex<X>&);
    template<class X> constexpr complex& operator/=(const complex<X>&);

};

29.5.4 complex member functions

template<class T> constexpr complex(T r = T(), T i = T());

1 Effects: Constructs an object of class complex.
Postconditions: \( \text{real}() == \text{re} \&\& \text{imag}() == \text{im} \).

```cpp
constexpr T real() const;
```

Returns: The value of the real component.

```cpp
constexpr void real(T val);
```

Effects: Assigns \( \text{val} \) to the real component.

```cpp
constexpr T imag() const;
```

Returns: The value of the imaginary component.

```cpp
constexpr void imag(T val);
```

Effects: Assigns \( \text{val} \) to the imaginary component.

### 29.5.5 complex member operators

#### [complex.member.ops]

```cpp
constexpr complex& operator+=(const T& rhs);
```

Effects: Adds the scalar value \( \text{rhs} \) to the real part of the complex value \(*\text{this}\) and stores the result in the real part of \(*\text{this}\), leaving the imaginary part unchanged.

Returns: \(*\text{this}\).

```cpp
constexpr complex& operator-=(const T& rhs);
```

Effects: Subtracts the scalar value \( \text{rhs} \) from the real part of the complex value \(*\text{this}\) and stores the result in the real part of \(*\text{this}\), leaving the imaginary part unchanged.

Returns: \(*\text{this}\).

```cpp
constexpr complex& operator*=(const T& rhs);
```

Effects: Multiplies the scalar value \( \text{rhs} \) by the complex value \(*\text{this}\) and stores the product in \(*\text{this}\).

Returns: \(*\text{this}\).

```cpp
constexpr complex& operator/=(const T& rhs);
```

Effects: Divides the scalar value \( \text{rhs} \) into the complex value \(*\text{this}\) and stores the quotient in \(*\text{this}\).

Returns: \(*\text{this}\).

```cpp
template<class X> constexpr complex& operator+=(const complex<X>& rhs);
```

Effects: Adds the complex value \( \text{rhs} \) to the complex value \(*\text{this}\) and stores the sum in \(*\text{this}\).

Returns: \(*\text{this}\).

```cpp
template<class X> constexpr complex& operator-=(const complex<X>& rhs);
```

Effects: Subtracts the complex value \( \text{rhs} \) from the complex value \(*\text{this}\) and stores the difference in \(*\text{this}\).

Returns: \(*\text{this}\).

```cpp
template<class X> constexpr complex& operator*=(const complex<X>& rhs);
```

Effects: Multiplies the complex value \( \text{rhs} \) by the complex value \(*\text{this}\) and stores the product in \(*\text{this}\).

Returns: \(*\text{this}\).

```cpp
template<class X> constexpr complex& operator/=(const complex<X>& rhs);
```

Effects: Divides the complex value \( \text{rhs} \) into the complex value \(*\text{this}\) and stores the quotient in \(*\text{this}\).

Returns: \(*\text{this}\).

### 29.5.6 complex non-member operations

#### [complex.ops]

```cpp
template<class T> constexpr complex<T> operator+(const complex<T>& lhs);
```

Returns: \( \text{complex}<\text{T}>(\text{lhs}) \).

Remarks: unary operator.
template<class T> constexpr complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
  Returns: complex<T>(lhs) += rhs.

template<class T> constexpr complex<T> operator+(const complex<T>& lhs, const T& rhs);
  4 Returns: complex<T>(-lhs.real(), -lhs.imag()).
  Remarks: unary operator.

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);
  6 Returns: complex<T>(-lhs.real(), -lhs.imag()).

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);
  Returns: complex<T>(lhs) -= rhs.

template<class T> constexpr complex<T> operator-(const T& lhs, const complex<T>& rhs);
  Returns: complex<T>(lhs) -= rhs.

template<class T> constexpr complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T> constexpr complex<T> operator*(const T& lhs, const complex<T>& rhs);
  Returns: complex<T>(lhs) *= rhs.

template<class T> constexpr complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T> constexpr complex<T> operator/(const T& lhs, const complex<T>& rhs);
  Returns: complex<T>(lhs) /= rhs.

template<class T> constexpr bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T> constexpr bool operator==(const complex<T>& lhs, const T& rhs);
  Returns: lhs.real() == rhs.real() && lhs.imag() == rhs.imag().
  Remarks: The imaginary part is assumed to be T(), or 0.0, for the T arguments.

template<class T> constexpr bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
  Returns: rhs.real() != lhs.real() || rhs.imag() != lhs.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 (30.7.4.2).
  If bad input is encountered, calls is.setstate(ios_base::failbit) (which may throw ios::failure
            (30.5.5.4)).
  Returns: is.
  Remarks: This extraction is performed as a series of simpler extractions. Therefore, the skipping of
            whitespaces 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, complex<T>& x);
  Effects: Inserts the complex number x onto the stream o as if it were implemented as follows:
            basic_ostringstream<charT, traits> s;
            s.flags(o.flags());
            s.imbue(o.getloc());
            s.precision(o.precision());
            s << '(' << x.real() << ',' << x.imag() << ')';
            return o << s.str();
  [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]

29.5.7 complex value operations

[complex.value.ops]

template<class T> constexpr T real(const complex<T>& x);
1
   Returns: x.real().

template<class T> constexpr T imag(const complex<T>& x);
2
   Returns: x.imag().

template<class T> constexpr T abs(const complex<T>& x);
3
   Returns: The magnitude of x.

template<class T> constexpr T arg(const complex<T>& x);
4
   Returns: The phase angle of x, or atan2(imag(x), real(x)).

template<class T> constexpr T norm(const complex<T>& x);
5
   Returns: The squared magnitude of x.

template<class T> constexpr complex<T> conj(const complex<T>& x);
6
   Returns: The complex conjugate of x.

template<class T> complex<T> proj(const complex<T>& x);
7
   Returns: The projection of x onto the Riemann sphere.
8
   Remarks: Behaves the same as the C function cproj. See also: ISO C 7.3.9.5

template<class T> complex<T> polar(const T& rho, const T& theta = T());
9
   Requires: rho shall be non-negative and non-NaN. theta shall be finite.
10
   Returns: The complex value corresponding to a complex number whose magnitude is rho and whose phase angle is theta.

29.5.8 complex transcendentals

[complex.transcendentals]

template<class T> complex<T> acos(const complex<T>& x);
1
   Returns: The complex arc cosine of x.
2
   Remarks: Behaves the same as the C function cacos. See also: ISO C 7.3.5.1

template<class T> complex<T> asin(const complex<T>& x);
3
   Returns: The complex arc sine of x.
4
   Remarks: Behaves the same as the C function casin. See also: ISO C 7.3.5.2

template<class T> complex<T> atan(const complex<T>& x);
5
   Returns: The complex arc tangent of x.
6
   Remarks: Behaves the same as the C function catan. See also: ISO C 7.3.5.3

template<class T> complex<T> acosh(const complex<T>& x);
7
   Returns: The complex arc hyperbolic cosine of x.
8
   Remarks: Behaves the same as the C function cacosh. See also: ISO C 7.3.6.1

template<class T> complex<T> asinh(const complex<T>& x);
9
   Returns: The complex arc hyperbolic sine of x.
10
   Remarks: Behaves the same as the C function casinh. See also: ISO C 7.3.6.2

template<class T> complex<T> atanh(const complex<T>& x);
11
   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

\begin{verbatim}
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 \(x\), \texttt{imag(log(x))} lies in the interval \([-\pi, \pi]\). [\textit{Note:} The semantics of this function are intended to be the same in C++ as they are for \texttt{clog} in C. — end note]
  Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> log10(const complex<T>& x);
  Returns: The complex common (base-10) logarithm of \(x\), defined as \texttt{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);
  Returns: The complex power of base \(x\) raised to the \(y\)th power, defined as \texttt{exp(y * 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. [\textit{Note:} The semantics of this function are intended to be the same in C++ as they are for \texttt{csqrt} in C. — end note]
  Remarks: The branch cuts are along the negative real axis.

template<class T> complex<T> tan(const complex<T>& x);
  Returns: The complex tangent of \(x\).

template<class T> complex<T> tanh(const complex<T>& x);
  Returns: The complex hyperbolic tangent of \(x\).
\end{verbatim}

\section*{29.5.9 Additional overloads [cmplx.over]}

The following function templates shall have additional overloads:

\begin{verbatim}
arg
norm
conj
proj
imag
real
\end{verbatim}

where \texttt{norm}, \texttt{conj}, \texttt{imag}, and \texttt{real} are \texttt{constexpr} overloads.

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>}.
\end{enumerate}
Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

- If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
- Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
- Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

### 29.5.10 Suffixes for complex number literals

This subclause describes literal suffixes for constructing complex number literals. The suffixes `i`, `il`, and `if` create complex numbers of the types `complex<double>`, `complex<long double>`, and `complex<float>` respectively, with their imaginary part denoted by the given literal number and the real part being zero.

```cpp
constexpr complex<long double> operator"il"(long double d);
constexpr complex<long double> operator"il"(unsigned long long d);
```

Returns: `complex<long double>{0.0L, static_cast<long double>(d)}`.

```cpp
constexpr complex<double> operator"i"(long double d);
constexpr complex<double> operator"i"(unsigned long long d);
```

Returns: `complex<double>{0.0, static_cast<double>(d)}`.

```cpp
constexpr complex<float> operator"if"(long double d);
constexpr complex<float> operator"if"(unsigned long long d);
```

Returns: `complex<float>{0.0f, static_cast<float>(d)}`.

### 29.6 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 `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:

- as `boolean` or equivalently as `boolean-valued`, if \( T \) is `bool`;
- otherwise as `integral` or equivalently as `integer-valued`, if `numeric_limits<T>::is_integer` is `true`;
- otherwise as `floating` or equivalently as `real-valued`.

If integer-valued, an entity may optionally be further characterized as `signed` or `unsigned`, according to `numeric_limits<T>::is_signed`.

Unless otherwise specified, all descriptions of calculations in 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_w` denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo \( 2^w \).
29.6.1 Requirements

29.6.1.1 General requirements

1 Throughout this subclause 29.6, the effect of instantiating a template:
   a) 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 (29.6.1.2).
   b) 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 (29.6.1.3).
   c) 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 (29.6.1.4).
   d) 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.
   e) 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.
   f) 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 29.6, 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 29.6, any constructor that can be called with a single argument and that satisfies a requirement specified in this subclause shall be declared explicit.

29.6.1.2 Seed sequence requirements

1 A seed sequence is an object that consumes a sequence of integer-valued data and produces a requested number of unsigned integer values \( i, 0 \leq i < 2^{32} \), based on the consumed data. [Note: Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful, for example, in applications requiring large numbers of random number engines. — end note]

2 A class S satisfies the requirements of a seed sequence if the expressions shown in Table 102 are valid and have the indicated semantics, and if S also satisfies all other requirements of this subclause 29.6.1.2. In that Table and throughout this subclause:
   a) T is the type named by S’s associated result_type;
   b) q is a value of S and r is a possibly const value of S;
   c) ib and ie are input iterators with an unsigned integer value_type of at least 32 bits;
   d) rb and re are mutable random access iterators with an unsigned integer value_type of at least 32 bits;
   e) ob is an output iterator; and
   f) il is a value of initializer_list<T>.

<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>
</tbody>
</table>
Table 102 — 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>S(il)</td>
<td>Same as S(il.begin(), il.end()).</td>
<td>same as S(il.begin(), il.end()).</td>
<td></td>
</tr>
<tr>
<td>q.generate(rb,re)</td>
<td>void</td>
<td>Does nothing if rb == re. Otherwise, fills the supplied sequence [rb,re] with 32-bit quantities that depend on the sequence supplied to the constructor and possibly also depend on the history of generate's previous invocations.</td>
<td>$\mathcal{O}(re - rb)$</td>
</tr>
<tr>
<td>r.size()</td>
<td>size_t</td>
<td>The number of 32-bit units that would be copied by a call to r.param.</td>
<td>constant</td>
</tr>
<tr>
<td>r.param(ob)</td>
<td>void</td>
<td>Copies to the given destination a sequence of 32-bit units that can be provided to the constructor of a second object of type S, and that would reproduce in that second object a state indistinguishable from the state of the first object.</td>
<td>$\mathcal{O}(r.size())$</td>
</tr>
</tbody>
</table>

29.6.1.3 Uniform random bit generator requirements

1 A uniform random bit generator $g$ of type $G$ is a function object returning unsigned integer values such that each value in the range of possible results has (ideally) equal probability of being returned. [Note: The degree to which $g$'s results approximate the ideal is often determined statistically. — end note]

2 A class $G$ satisfies the requirements of a uniform random bit generator if the expressions shown in Table 103 are valid and have the indicated semantics, and if $G$ also satisfies all other requirements of this subclause 29.6.1.3. In that Table and throughout this subclause:

a) $T$ is the type named by $G$'s associated result_type, and
b) $g$ is a value of $G$.

Table 103 — Uniform random bit generator requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>G::result_type</td>
<td>T</td>
<td>T is an unsigned integer type (6.7.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>g()</td>
<td>T</td>
<td>Returns a value in the closed interval [$G::\text{min}(), G::\text{max}()$].</td>
<td>amortized constant</td>
</tr>
<tr>
<td>G::min()</td>
<td>T</td>
<td>Denotes the least value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
<tr>
<td>G::max()</td>
<td>T</td>
<td>Denotes the greatest value potentially returned by operator().</td>
<td>compile-time</td>
</tr>
</tbody>
</table>

3 The following relation shall hold: $G::\text{min}() < G::\text{max}()$.

29.6.1.4 Random number engine requirements

1 A random number engine (commonly shortened to engine) $e$ of type $E$ is a uniform random bit generator that additionally meets the requirements (e.g., for seeding and for input/output) specified in this subclause.
At any given time, $e$ has a state $e_i$ for some integer $i \geq 0$. Upon construction, $e$ has an initial state $e_0$. An engine’s state may be established via a constructor, a `seed` function, assignment, or a suitable `operator>>`.

E’s specification shall define:

a) the size of E’s state in multiples of the size of `result_type`, given as an integral constant expression;

b) the transition algorithm $TA$ by which $e$’s state $e_i$ is advanced to its successor state $e_{i+1}$; and

c) the generation algorithm $GA$ by which an engine’s state is mapped to a value of type `result_type`.

A class $E$ that satisfies the requirements of a uniform random bit generator (29.6.1.3) also satisfies the requirements of a random number engine if the expressions shown in Table 104 are valid and have the indicated semantics, and if $E$ also satisfies all other requirements of this subclause 29.6.1.4. In that Table and throughout this subclause:

- $T$ is the type named by $E$’s associated `result_type`;
- $e$ is a value of $E$, $v$ is an lvalue of $E$, $x$ and $y$ are (possibly `const`) values of $E$;
- $s$ is a value of $T$;
- $q$ is an lvalue satisfying the requirements of a seed sequence (29.6.1.2);
- $z$ is a value of type `unsigned long long`;
- $os$ is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and
- $is$ is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to Clause 24 and Clause 30.

Table 104 — Random number engine requirements

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E()$</td>
<td></td>
<td>Creates an engine with the same initial state as all other default-constructed engines of type $E$.</td>
<td>$O$(size of state)</td>
</tr>
<tr>
<td>$E(x)$</td>
<td></td>
<td>Creates an engine that compares equal to $x$.</td>
<td>$O$(size of state)</td>
</tr>
<tr>
<td>$E(s)$</td>
<td></td>
<td>Creates an engine with initial state determined by $s$.</td>
<td>$O$(size of state)</td>
</tr>
<tr>
<td>$E(q)$</td>
<td>void</td>
<td>Creates an engine with an initial state that depends on a sequence produced by one call to $q.generate$.</td>
<td>same as complexity of $q.generate$ called on a sequence whose length is size of state</td>
</tr>
<tr>
<td>$e.seed()$</td>
<td>void</td>
<td>Postconditions: $e == E()$.</td>
<td>same as $E()$</td>
</tr>
<tr>
<td>$e.seed(s)$</td>
<td>void</td>
<td>Postconditions: $e == E(s)$.</td>
<td>same as $E(s)$</td>
</tr>
<tr>
<td>$e.seed(q)$</td>
<td>void</td>
<td>Postconditions: $e == E(q)$.</td>
<td>same as $E(q)$</td>
</tr>
<tr>
<td>$e()$</td>
<td>$T$</td>
<td>Advances $e$’s state $e_i$ to $e_{i+1} = TA(e_i)$ and returns $GA(e_i)$.</td>
<td>per Table 103</td>
</tr>
<tr>
<td>$e.discard(z)$</td>
<td>void</td>
<td>Advances $e$’s state $e_i$ to $e_{i+z}$ by any means equivalent to $z$ consecutive calls $e()$.</td>
<td>no worse than the complexity of $z$ consecutive calls $e()$</td>
</tr>
</tbody>
</table>

---

271) This constructor (as well as the subsequent corresponding `seed()` function) may be particularly useful to applications requiring a large number of independent random sequences.

272) 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 $e()$. 

§ 29.6.1.4
<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>x == y</td>
<td>bool</td>
<td>This operator is an equivalence relation. With $S_x$ and $S_y$ as the infinite sequences of values that would be generated by repeated future calls to $x()$ and $y()$, respectively, returns true if $S_x = S_y$; else returns false.</td>
<td>$O(\text{size of state})$</td>
</tr>
<tr>
<td>x != y</td>
<td>bool</td>
<td>$!(x == y)$</td>
<td>$O(\text{size of state})$</td>
</tr>
</tbody>
</table>
| os << x    | reference to the type of os | With os.
|            |             | fmtflags set to ios_.base::dec|ios_base::left and the fill character set to the space character, writes to os the textual representation of x's current state. In the output, adjacent numbers are separated by one or more space characters. Postconditions: The os.
|            |             | fmtflags and fill character are unchanged. | $O(\text{size of state})$ |
| is >> v    | reference to the type of is | With is.
|            |             | fmtflags set to ios_base::dec, sets v's state as determined by reading its textual representation from is. If bad input is encountered, ensures that v's state is unchanged by the operation and calls is.setstate(ios::failbit) (which may throw ios::failure (30.5.5.4)). If a textual representation written via os << x was subsequently read via is >> v, then x == v provided that there have been no intervening invocations of x or of v. Requires: is provides a textual representation that was previously written using an output stream whose imbued locale was the same as that of is, and whose type's template specialization arguments charT and traits were respectively the same as those of is. Postconditions: The is.
|            |             | fmtflags are unchanged. | $O(\text{size of state})$ |

5 E shall meet the requirements of CopyConstructible (Table 24) and CopyAssignable (Table 26) types. These operations shall each be of complexity no worse than $O(\text{size of state})$.

29.6.1.5 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.
The requirements of a random number engine type shall be interpreted as follows with respect to a random number engine adaptor type.

`A::A();`

**Effects:** The base engine is initialized as if by its default constructor.

`bool operator==(const A& a1, const A& a2);`

**Returns:** `true` if a1's base engine is equal to a2's base engine. Otherwise returns `false`.

`A::A(result_type s);`

**Effects:** The base engine is initialized with s.

`template<class Sseq> A::A(Sseq& q);`

**Effects:** The base engine is initialized with q.

`void seed();`

**Effects:** With b as the base engine, invokes b.seed().

`void seed(result_type s);`

**Effects:** With b as the base engine, invokes b.seed(s).

`template<class Sseq> void seed(Sseq& q);`

**Effects:** With b as the base engine, invokes b.seed(q).

A shall also satisfy the following additional requirements:

a) The complexity of each function shall not exceed the complexity of the corresponding function applied to the base engine.

b) The state of A shall include the state of its base engine. The size of A's state shall be no less than the size of the base engine.

c) Copying A's state (e.g., during copy construction or copy assignment) shall include copying the state of the base engine of A.

d) The textual representation of A shall include the textual representation of its base engine.

29.6.1.6 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). A distribution's specification identifies its associated probability function p(z) or P(z).

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 105 are valid and have the indicated semantics, and if D and its associated types also satisfy all other requirements of this subclause 29.6.1.6. In that Table and throughout this subclause,

a) T is the type named by D's associated `result_type`;

b) P is the type named by D's associated `param_type`;

c) d is a value of D, and x and y are (possibly `const`) values of D;

d) `glb` and `lub` are values of T respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by d's `operator()`, as determined by the current values of d's parameters;

e) p is a (possibly `const`) value of P;

f) g, g1, and g2 are lvalues of a type satisfying the requirements of a uniform random bit generator (29.6.1.3);

g) os is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and
h) 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 24 and Clause 30.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Pre/post-condition</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>D::result_type</td>
<td>T</td>
<td>T is an arithmetic type (6.7.1).</td>
<td>compile-time</td>
</tr>
<tr>
<td>D::param_type</td>
<td>P</td>
<td></td>
<td>compile-time</td>
</tr>
<tr>
<td>D()</td>
<td></td>
<td>Creates 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>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>Postconditions: 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</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</td>
</tr>
<tr>
<td>x.min()</td>
<td>T</td>
<td>Returns glb.</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 S1 = S2, where S1 and S2 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>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>os &lt;&lt; x</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. <em>Postconditions:</em> The os.fmtflags and fill character are unchanged.</td>
<td></td>
</tr>
<tr>
<td>is &gt;&gt; d</td>
<td>reference to the type of is</td>
<td>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 (30.5.5.4)). <em>Requires:</em> 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. <em>Postconditions:</em> The is.fmtflags are unchanged.</td>
<td></td>
</tr>
</tbody>
</table>

4 D shall satisfy the requirements of CopyConstructible (Table 24) and CopyAssignable (Table 26) types.

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 29.6, declarations of D::param_type are in the form of typedefs for convenience of exposition only.

8 P shall satisfy the requirements of CopyConstructible (Table 24), CopyAssignable (Table 26), and EqualityComparable (Table 20) types.

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

```cpp
using distribution_type = D;
```

29.6.2 Header `<random>` synopsis

```cpp
#include <initializer_list>

namespace std {
    // 29.6.3.1, class template linear_congruential_engine
    template<class UIntType, UIntType a, UIntType c, UIntType m>
        class linear_congruential_engine;
}
```
// 29.6.3.2, class template mersenne_twister_engine
template<class UIntType, size_t w, size_t n, size_t m, size_t r,
        UIntType a, size_t u, UIntType d, size_t s,
        UIntType b, size_t t,
        UIntType c, size_t l, UIntType f>
class mersenne_twister_engine;

// 29.6.3.3, class template subtract_with_carry_engine
template<class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine;

// 29.6.4.2, class template discard_block_engine
template<class Engine, size_t p, size_t r>
class discard_block_engine;

// 29.6.4.3, class template independent_bits_engine
template<class Engine, size_t w, class UIntType>
class independent_bits_engine;

// 29.6.4.4, class template shuffle_order_engine
template<class Engine, size_t k>
class shuffle_order_engine;

// 29.6.5, 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;

// 29.6.6, class random_device
class random_device;

// 29.6.7.1, class seed_seq
class seed_seq;

// 29.6.7.2, function template generate_canonical
template<class RealType, size_t bits, class URBG>
RealType generate_canonical(URBG& g);

// 29.6.8.2.1, class template uniform_int_distribution
template<class IntType = int>
class uniform_int_distribution;

// 29.6.8.2.2, class template uniform_real_distribution
template<class RealType = double>
class uniform_real_distribution;

// 29.6.8.3.1, class bernoulli_distribution
class bernoulli_distribution;

// 29.6.8.3.2, class template binomial_distribution
template<class IntType = int>
class binomial_distribution;

// 29.6.8.3.3, class template geometric_distribution
template<class IntType = int>
class geometric_distribution;
negative_binomial_distribution

template<class IntType = int>
class negative_binomial_distribution;

poisson_distribution

template<class IntType = int>
class poisson_distribution;

exponential_distribution

template<class RealType = double>
class exponential_distribution;

gamma_distribution

template<class RealType = double>
class gamma_distribution;

weibull_distribution

template<class RealType = double>
class weibull_distribution;

extreme_value_distribution

template<class RealType = double>
class extreme_value_distribution;

normal_distribution

template<class RealType = double>
class normal_distribution;

lognormal_distribution

template<class RealType = double>
class lognormal_distribution;

chi_squared_distribution

template<class RealType = double>
class chi_squared_distribution;

cauchy_distribution

template<class RealType = double>
class cauchy_distribution;

fisher_f_distribution

template<class RealType = double>
class fisher_f_distribution;

student_t_distribution

template<class RealType = double>
class student_t_distribution;

discrete_distribution

template<class IntType = int>
class discrete_distribution;

piecewise_constant_distribution

template<class RealType = double>
class piecewise_constant_distribution;

piecewise_linear_distribution

template<class RealType = double>
class piecewise_linear_distribution;
29.6.3 Random number engine class templates

Each type instantiated from a class template specified in this subclause 29.6.3 satisfies the requirements of a random number engine (29.6.1.4) type.

Except where specified otherwise, the complexity of each function specified in this subclause 29.6.3 is constant.

Except where specified otherwise, no function described in this subclause 29.6.3 throws an exception.

Every function described in this subclause 29.6.3 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 29.6.3 only for engine operations that are not described in 29.6.1.4 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 29.6.3 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 (29.6.3) and in subclause 29.6.4:

1. if the constructor
   
   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;

2. if the member function
   
   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.

29.6.3.1 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(result_type s = default_seed);
    template<class Sseq> explicit linear_congruential_engine(Sseq& q);
    void seed(result_type s = default_seed);
    template<class Sseq> void seed(Sseq& q);
};
```
The generation algorithm determines the unsigned integer values
of the template parameter \( m \) used throughout this subclause 29.6.3.1 is numeric_limits<result_type>::max() plus 1. [Note: \( m \) need not be representable as a value of type result_type.  
—end note]

If the template parameter \( m \) is not 0, the following relations shall hold: \( a < m \) and \( c < m \).

The textual representation consists of the value of \( x_i \).

**Effects:** 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 \( k = \left \lceil \log x_\text{max} m \right \rceil \) and \( a \) an array (or equivalent) of length \( k + 3 \), invokes \( q \).generate\((a + 0, a + k + 3)\) and then computes \( S = \left( \sum_{j=0}^{k-1} a_j 3^j \cdot 2^{\ell j} \right) \mod m \). If \( c \mod m \) is 0 and \( S \) is 0, sets the engine’s state to 1, else sets the engine’s state to \( S \).

### 29.6.3.2 Class template mersenne_twister_engine

**A mersenne_twister_engine** random number engine produces unsigned integer random numbers in the closed interval \([0, 2^{32} - 1]\). The state \( X \) 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:

- 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 \( Y \).
- 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:

- Let \( z_1 = X_i \text{ xor } \left( (X_i \text{ rshift } u) \text{ bitand } d \right) \).
- Let \( z_2 = z_1 \text{ xor } \left( (z_1 \text{ lshift } s) \text{ bitand } b \right) \).
- Let \( z_3 = z_2 \text{ xor } \left( (z_2 \text{ lshift } t) \text{ bitand } c \right) \).
- Let \( z_4 = z_3 \text{ xor } (z_3 \text{ rshift } \ell) \).

**template<class UIntType, size_t w, size_t n, size_t m, size_t r, 
UIntType a, size_t u, UIntType d, size_t s, 
UIntType b, size_t t, 
UIntType c, size_t l, UIntType f>**

**class mersenne_twister_engine {**

**public:**

```cpp
  using result_type = UIntType;
```

```cpp
  // generating functions
  result_type operator()();
  void discard(unsigned long long z);
};
```

273) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.
The generation algorithm is given by

\[
\text{state } A_{1-n} \text{ transition is performed as follows:}
\]

\[
\begin{align*}
\text{The state } x_i \text{ consists of the values of } X_{i-n}, \ldots, X_{i-1}, \text{ in that order.}
\end{align*}
\]

4 The following relations shall hold: 0 < \( m, m \leq n \), 2\( u < w \), \( r < w \), \( u < w \), \( s < w \), \( t < w \), \( l \leq w \), \( w \leq \text{numeric_limits<UIntType>::digits} \), \( a \leq (1u<<w) - 1u \), \( b \leq (1u<<w) - 1u \), \( c \leq (1u<<w) - 1u \), \( d \leq (1u<<w) - 1u \), and \( f \leq (1u<<w) - 1u \).

5 The textual representation of \( x_i \) consists of the values of \( X_{i-n}, \ldots, X_{i-1} \), in that order.

6 **Effects:** Constructs a \texttt{mersenne_twister_engine} object. Sets \( X_{-n} \) to value mod \( 2^w \). Then, iteratively for \( i = 1-n, \ldots, -1 \), sets \( X_i \) to

\[
[j \cdot (X_{i-1} \oplus (X_{i-1} \text{ rshift}(w-2))) + i \text{ mod } n] \mod 2^w.
\]

7 **Complexity:** \( O(n) \).

8 **Effects:** Constructs a \texttt{mersenne_twister_engine} object. With \( k = \lceil w/32 \rceil \) and \( a \) an array (or equivalent) of length \( n \cdot k \), invokes \( q \).generate \((a+0, a+n \cdot k) \) and then, iteratively for \( i = -n, \ldots, -1 \), sets \( X_i \) to \( \left( \sum_{j=0}^{k-1} d_k(i+n) + j \cdot 2^{32j} \right) \mod 2^w \). Finally, if the most significant \( w - r \) bits of \( X_{-n} \) are zero, and if each of the other resulting \( X_i \) is 0, changes \( X_{-n} \) to \( 2^{w-1} \).

29.6.3.3 **Class template subtract_with_carry_engine**

A \texttt{subtract_with_carry_engine} random number engine produces unsigned integer random numbers.

2 The state \( x_i \) of a \texttt{subtract_with_carry_engine} object \( x \) is of size \( \Theta(r) \), and consists of a sequence \( X \) of \( r \) integer values \( 0 \leq X_j < m = 2^w \); 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.

3 The state transition is performed as follows:

a) Let \( Y = X_{i-r} - X_{i-r} - c \).

b) Set \( X_i \) to \( y \equiv Y \mod m \). 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 \( m^r - m^s + 1 \) and \( a = b - (b - 1)/m \). — end note]

4 The generation algorithm is given by \( GA(x_i) = y \), where \( y \) is the value produced as a result of advancing the engine’s state as described above.
template<class UIntType, size_t w, size_t s, size_t r>
class subtract_with_carry_engine {
public:
    // types
    using result_type = UIntType;
    // engine characteristics
    static constexpr size_t word_size = w;
    static constexpr size_t short_lag = s;
    static constexpr size_t long_lag = r;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return m - 1; }
    static constexpr result_type default_seed = 19780503u;
    // constructors and seeding functions
    explicit subtract_with_carry_engine(result_type value = default_seed);
    template<class Sseq> explicit subtract_with_carry_engine(Sseq& q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq& q);
    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
};

The following relations shall hold: 0u < s, s < r, 0 < w, and w <= numeric_limits<UIntType>::digits.
The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by c.

explicit subtract_with_carry_engine(result_type value = default_seed);

Effects: Constructs a subtract_with_carry_engine object. Sets the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, as specified below. If $X_{i-1}$ is then 0, sets c to 1; otherwise sets c to 0.

To set the values $X_k$, first construct e, a linear_congruential_engine object, as if by the following definition:

linear_congruential_engine<result_type, 40014u, 0u, 2147483563u> e(value == 0u ? default_seed : value);

Then, to set each $X_k$, obtain new values $z_0, \ldots, z_{n-1}$ from $n = \lceil w/32 \rceil$ successive invocations of e taken modulo $2^{32}$. Set $X_k$ to $\left( \sum_{j=0}^{n-1} z_j \cdot 2^{32j} \right) \pmod{m}$. If $X_{i-1}$ is then 0, sets c to 1; otherwise sets c to 0.

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 $\left( \sum_{j=0}^{k-1} a_k(i+r)+j \cdot 2^{32j} \right) \pmod{m}$. If $X_{i-1}$ is then 0, sets c to 1; otherwise sets c to 0.

29.6.4 Random number engine adaptor class templates

In general

Each type instantiated from a class template specified in this subclause 29.6.4 satisfies the requirements of a random number engine adaptor (29.6.1.5) type.

Except where specified otherwise, the complexity of each function specified in this subclause 29.6.4 is constant.

Except where specified otherwise, no function described in this subclause 29.6.4 throws an exception.

Every function described in this subclause 29.6.4 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 29.6.4 only for adaptor operations that are not described in subclause 29.6.1.5 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 29.6.4 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.

29.6.4.2 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:
    // types
    using result_type = typename Engine::result_type;
    // engine characteristics
    static constexpr size_t block_size = p;
    static constexpr size_t used_block = r;
    static constexpr result_type min() { return Engine::min(); }
    static constexpr result_type max() { return Engine::max(); }
    // constructors and seeding functions
    discard_block_engine();
    explicit discard_block_engine(const Engine& e);
    explicit discard_block_engine(Engine&& e);
    explicit discard_block_engine(result_type s);
    template<class Sseq> explicit discard_block_engine(Sseq& q);
    void seed();
    void seed(result_type s);
    template<class Sseq> void seed(Sseq& q);
    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
    // property functions
    const Engine& base() const noexcept { return e; }
private:
    Engine e; // exposition only
    int n; // exposition only
};
```

The following relations shall hold: \( 0 < r \) and \( r \leq p \).

The textual representation consists of the textual representation of \( e \) followed by the value of \( n \).

In addition to its behavior pursuant to subclause 29.6.1.5, each constructor that is not a copy constructor sets \( n \) to 0.

29.6.4.3 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:
a) Let $R = e.\text{max}() - e.\text{min}() + 1$ and $m = \lceil \log_2 R \rceil$.

b) With $n$ as determined below, let $w_0 = \lfloor w/n \rfloor$, $n_0 = n - w \mod n$, $y_0 = 2^{w_0} \lceil R/2^{w_0} \rceil$, and $y_1 = 2^{w_0+1} \lceil R/2^{w_0+1} \rceil$.

c) Let $n = \lfloor w/m \rfloor$ if and only if the relation $R - y_0 \leq \lfloor y_0/n \rfloor$ holds as a result. Otherwise let $n = 1 + \lfloor w/m \rfloor$. 

[Note: The relation $w = n_0 w_0 + (n - n_0)(w_0 + 1)$ always holds. —end note]

3 The transition algorithm is carried out by invoking $e()$ as often as needed to obtain $n_0$ values less than $y_0 + e.\text{min}()$ and $n - n_0$ values less than $y_1 + e.\text{min}()$.

4 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 ($k = 0; k < n_0; k += 1$) {
    do $u = e() - e.\text{min}();$ while ($u \geq y_0$);
    $S = 2^{w_0} \cdot S + u \mod 2^{w_0}$;
}

for ($k = n_0; k < n; k += 1$) {
    do $u = e() - e.\text{min}();$ while ($u \geq y_1$);
    $S = 2^{w_0+1} \cdot S + u \mod 2^{w_0+1}$;
}

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
};

5 The following relations shall hold: $0 < w$ and $w \leq \text{numeric\_limits<result\_type>::digits}$.

6 The textual representation consists of the textual representation of $e$.

29.6.4.4 Class template shuffle_order_engine

A shuffle_order_engine random number engine adaptor produces the same random numbers that are produced by some base engine $e$, but delivers them in a different sequence. The state $x_i$ of a shuffle_order_engine engine adaptor object $x$ consists of the state $e_i$ of its base engine $e$, an additional value $Y$ of the type delivered by $e$, and an additional sequence $V$ of $k$ values also of the type delivered by $e$. The size of the state is the size of $e$'s state plus $k + 1$.

The transition algorithm permutes the values produced by $e$. The state transition is performed as follows:
a) Calculate an integer \( j = \left\lfloor k \left( Y - e_{\text{min}} \right) \over e_{\text{max}} - e_{\text{min}} + 1 \right\rfloor \).

b) Set \( Y \) to \( V_j \) and then set \( V_j \) to \( e() \).

3 The generation algorithm yields the last value of \( Y \) produced while advancing \( e \)'s state as described above.

```cpp
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 29.6.1.5, each constructor that is not a copy constructor initializes \( V[0] \), \ldots, \( V[k-1] \) and \( Y \), in that order, with values returned by successive invocations of \( e() \).

### 29.6.5 Engines and engine adaptors with predefined parameters [rand.predef]

1 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `minstd_rand0` shall produce the value 1043618065.

```cpp
using minstd_rand0 =
    linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>;
```

2 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `minstd_rand` shall produce the value 399268537.

```cpp
using minstd_rand =
    linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>;
```

3 **Required behavior:** The 10000th consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

```cpp
using mt19937 =
    mersenne_twister_engine<uint_fast32_t,
        32, 624, 397, 31, 0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18, 1812433253>;
```

§ 29.6.5
using mt19937_64 =
   mersenne_twister_engine<
      uint_fast64_t,
      64,312,156,31,0xb5026f5a96619e9,29,
      0x5555555555555555,17,
      0x71d67ffeda60000,37,
      0xff7ee00000000000,43,
      6364136223846793005>;  

4 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 111239016.

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]

29.6.6 Class random_device  [rand.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
      explicit random_device(const string& token = implementation-defined);

      // 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 = implementation-defined);

Effects: Constructs a random_device nondeterministic uniform random bit generator object. The
semantics and default value of the token parameter are implementation-defined.\textsuperscript{274}

Throws: A value of an implementation-defined type derived from exception if the random_device
could not be initialized.

double entropy() const noexcept;

Returns: If the implementation employs a random number engine, returns 0.0. Otherwise, returns
an entropy estimate\textsuperscript{275} for the random numbers returned by operator(), in the range \min() to
\log_2(\max() + 1).

result_type operator()();

Returns: A nondeterministic random value, uniformly distributed between \min() and \max(), inclusive.
It is implementation-defined how these values are generated.

Throws: A value of an implementation-defined type derived from exception if a random number could
not be obtained.

29.6.7 Utilities \[rand'util\]
29.6.7.1 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
};

\textsuperscript{274} The parameter is intended to allow an implementation to differentiate between different sources of randomness.
\textsuperscript{275} If a device has \(n\) states whose respective probabilities are \(P_0, \ldots, P_{n-1}\), the device entropy \(S\) is defined as
\[ S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i. \]
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.
4  Effects: Same as seed_seq(il.begin(), il.end()).

template<class InputIterator>
seed_seq(InputIterator begin, InputIterator end);

5  Requires: InputIterator shall satisfy the requirements of an input iterator (Table 95) type. Moreover, iterator_traits<InputIterator>::value_type shall denote an integer type.
6  Effects: Constructs a seed_seq object by the following algorithm:
   for( InputIterator s = begin; s != end; ++s)
      v.push_back((*s) mod232);

template<class RandomAccessIterator>
void generate(RandomAccessIterator begin, RandomAccessIterator end);

7  Requires: RandomAccessIterator shall meet the requirements of a mutable random access iterator (27.2.7). Moreover, iterator_traits<RandomAccessIterator>::value_type shall denote an unsigned integer type capable of accommodating 32-bit quantities.
8  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 232, each indexing operator applied to begin is to be taken modulo n, and T(x) is defined as x xor (x rshift 27):
   a) 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
      \[ t = (n \geq 623) ? 11 : (n \geq 68) ? 7 : (n \geq 39) ? 5 : (n \geq 7) ? 3 : (n - 1)/2; \]
   b) 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
      \[ r_1 = 1664525 \cdot T(begin[k] \ xor \ begin[k + p] \ xor \ begin[k - 1]) \]
      \[ r_2 = r_1 + \begin{cases} s, & k = 0 \\ k \mod n + v[k - 1], & 0 < k \leq s \\ k \mod n, & s < k \end{cases} \]
      and, in order, increment begin[k + p] by r1, increment begin[k + q] by r2, and set begin[k] to r2.
   c) Transform the elements of the range again, beginning where the previous step ended: iteratively for \( k = m, \ldots, m + n - 1 \), calculate values
      \[ r_3 = 1566083941 \cdot T(begin[k] + begin[k + p] + begin[k - 1]) \]
      \[ r_4 = r_3 - (k \ mod n) \]
      and, in order, update begin[k + p] by xoring it with r3, update begin[k + q] by xoring it with r4, and set begin[k] to r4.
9  Throws: What and when RandomAccessIterator operations of begin and end throw.

size_t size() const noexcept;

10  Returns: The number of 32-bit units that would be returned by a call to param().
11  Complexity: Constant time.
template<class OutputIterator>
void param(OutputIterator dest) const;

Requires: OutputIterator shall satisfy the requirements of an output iterator (27.2.4). 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:
copy(v.begin(), v.end(), dest);

Throws: What and when OutputIterator operations of dest throw.

29.6.7.2 Function template generate_canonical [rand.util.canonical]
1 Each function instantiated from the template described in this subclause 29.6.7.2 maps the result of one or more invocations of a supplied uniform random bit generator g to one member of the specified 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]

template<class RealType, size_t bits, class URBG>
RealType generate_canonical(URBG& g);

Complexity: Exactly \( k = \) max(1, \left\lfloor \frac{b}{\log_2 R} \right\rfloor) invocations of g, where \( b^{276} \) is the lesser of numeric_limits<RealType>::digits and bits, and R is the value of g.max() − g.min() + 1.

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 RealType.

Returns: \( S/R^k \).

Throws: What and when g throws.

29.6.8 Random number distribution class templates [rand.dist]
29.6.8.1 In general [rand.dist.general]
1 Each type instantiated from a class template specified in this subclause 29.6.8 satisfies the requirements of a random number distribution (29.6.1.6) type.

2 Descriptions are provided in this subclause 29.6.8 only for distribution operations that are not described in 29.6.1.6 or for operations where there is additional semantic information. In particular, declarations for copy constructors, for copy assignment operators, for streaming operators, and for equality and inequality operators are not shown in the synopses.

3 The algorithms for producing each of the specified distributions are implementation-defined.

4 The value of each probability density function \( p(z) \) and of each discrete probability function \( P(z_i) \) specified in this subclause is 0 everywhere outside its stated domain.

29.6.8.2 Uniform distributions [rand.dist.uni]
29.6.8.2.1 Class template uniform_int_distribution [rand.dist.uni.int]
1 A uniform_int_distribution random number distribution produces random integers \( i, a \leq i \leq b \), distributed according to the constant discrete probability function
\[
P(i \mid a, b) = 1/(b - a + 1)
\]

276) \( b \) is introduced to avoid any attempt to produce more bits of randomness than can be held in RealType.
template<class IntType = int>
class uniform_int_distribution {
public:
  // types
  using result_type = IntType;
  using param_type = unspecified;
  // constructors and reset functions
  explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
  explicit uniform_int_distribution(const param_type& parm);
  void reset();
  // generating functions
  template<class URBG>
  result_type operator()(URBG& g);
  template<class URBG>
  result_type operator()(URBG& g, const param_type& parm);
  // 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 = 0, IntType b = numeric_limits<IntType>::max());

Requires: a ≤ b.
Effects: Constructs a uniform_int_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.

result_type b() const;
Returns: The value of the b parameter with which the object was constructed.

29.6.8.2.2 Class template uniform_real_distribution [rand.dist.uni.real]
A uniform_real_distribution random number distribution produces random numbers x, a ≤ x < b, distributed according to the constant probability density function

\[ p(x | a, b) = 1/(b - a) \text{.} \]

[ Note: This implies that p(x | a, b) is undefined when a == b. —end note ]

template<class RealType = double>
class uniform_real_distribution {
public:
  // types
  using result_type = RealType;
  using param_type = unspecified;
  // constructors and reset functions
  explicit uniform_real_distribution(RealType a = 0.0, 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);
};

§ 29.6.8.2.2
// property functions
result_type a() const;
result_type b() const;
param_type param() const;
void param(const param_type& parm);
result_type min() const;
result_type max() const;
}

explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);

Requires: a ≤ b and b − a ≤ numeric_limits<RealType>::max().

Effects: Constructs a 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.

result_type b() const;

Returns: The value of the b parameter with which the object was constructed.

29.6.8.3 Bernoulli distributions [rand.dist.bern]

29.6.8.3.1 Class bernoulli_distribution [rand.dist.bern.bernoulli]

A bernoulli_distribution random number distribution produces bool values b distributed according to the discrete probability function

\[
P(b|p) = \begin{cases} 
p & \text{if } b = \text{true} \\ 
1 - p & \text{if } b = \text{false} 
\end{cases}
\]

class bernoulli_distribution {
public:

    // types
    using result_type = bool;
    using param_type = unspecified;

    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
    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 = 0.5);

Requires: 0 ≤ p ≤ 1.

Effects: Constructs a 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.
29.6.8.3.2 Class template binomial_distribution [rand.dist.bern.bin]

A `binomial_distribution` random number distribution produces integer values \( i \geq 0 \) distributed according to the discrete probability function

\[
P(i \mid t, p) = \binom{t}{i} \cdot p^i \cdot (1 - p)^{t-i}.
\]

```cpp
template<class IntType = int>
class binomial_distribution {
  public:
  // types
  using result_type = IntType;
  using param_type = unspecified;

  // constructors and reset functions
  explicit binomial_distribution(IntType t = 1, 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;
};
```

1. Requires: \( 0 \leq p \leq 1 \) and \( 0 \leq t \).
2. Effects: Constructs a `binomial_distribution` object; \( t \) and \( p \) correspond to the respective parameters of the distribution.

```cpp
explicit binomial_distribution(IntType t = 1, double p = 0.5);
```

3. Returns: The value of the \( t \) parameter with which the object was constructed.

```cpp
double p() const;
```

4. Returns: The value of the \( p \) parameter with which the object was constructed.

29.6.8.3.3 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

\[
P(i \mid p) = p \cdot (1 - p)^i.
\]

```cpp
template<class IntType = int>
class geometric_distribution {
  public:
  // types
  using result_type = IntType;
  using param_type = unspecified;

  // constructors and reset functions
  explicit geometric_distribution(double p = 0.5);
  explicit geometric_distribution(const param_type& parm);
  void reset();

  // 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;
};
```

1. Requires: \( 0 \leq p \leq 1 \) and \( 0 \leq t \).
2. Effects: Constructs a `geometric_distribution` object; \( t \) and \( p \) correspond to the respective parameters of the distribution.

```cpp
explicit geometric_distribution(double p = 0.5);
```

3. Returns: The value of the \( t \) parameter with which the object was constructed.

```cpp
double p() const;
```

4. Returns: The value of the \( p \) parameter with which the object was constructed.
// generating functions
template<class URBG>
  result_type operator()(URBG& g);

// 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 = 0.5);

2 Requires: 0 < p < 1.
3 Effects: Constructs a geometric_distribution object; p corresponds to the parameter of the distribution.

double p() const;
4 Returns: The value of the p parameter with which the object was constructed.

29.6.8.3.4 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

\[
P(i|k,p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i.\]

[Note: This implies that \( P(i|k,p) \) is undefined when \( p == 1. \) — end note]

template<class IntType = int>
  class negative_binomial_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type = unspecified;

    // constructor and reset functions
    explicit negative_binomial_distribution(IntType k = 1, 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 = 1, double p = 0.5);

2 Requires: 0 < p ≤ 1 and 0 < k.
3 Effects: Constructs a negative_binomial_distribution object; k and p correspond to the respective parameters of the distribution.

§ 29.6.8.3.4
IntType k() const;

Returns: The value of the k parameter with which the object was constructed.

double p() const;

Returns: The value of the p parameter with which the object was constructed.

### 29.6.8.4 Poisson distributions

#### 29.6.8.4.1 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.

```cpp
template<class IntType = int>
class poisson_distribution {
public:
    // types
    using result_type = IntType;
    using param_type = unspecified;
    // constructors and reset functions
    explicit poisson_distribution(double mean = 1.0);
    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;
};
```

#### 29.6.8.4.2 Class template exponential_distribution

An exponential_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | \lambda) = \lambda e^{-\lambda x}.
\]

```cpp
template<class RealType = double>
class exponential_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;
```
// constructors and reset functions
explicit exponential_distribution(RealType lambda = 1.0);
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 = 1.0);

Requires: 0 < lambda.
Effects: Constructs an exponential_distribution object; lambda corresponds to the parameter of the distribution.

RealType lambda() const;
Returns: The value of the lambda parameter with which the object was constructed.

29.6.8.4.3 Class template gamma_distribution
A gamma_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x | \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^{\alpha} \cdot \Gamma(\alpha)} \cdot x^{\alpha - 1}.
\]

template<class RealType = double>
class gamma_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructors and reset functions
    explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
    explicit gamma_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType alpha() const;
    RealType beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
explicit gamma_distribution(RealType alpha = 1.0, 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.

29.6.8.4.4 Class template weibull_distribution

A weibull_distribution random number distribution produces random numbers \(x \geq 0\) distributed according to the probability density function

\[
p(x \g g a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^a\right).\]

A weibull_distribution object; \(\alpha\) and \(\beta\) correspond to the respective parameters of the distribution.

RealType a() const;
Returns: The value of the \(\alpha\) parameter with which the object was constructed.

RealType b() const;
Returns: The value of the \(\beta\) parameter with which the object was constructed.
29.6.8.4.5 Class template extreme_value_distribution

An extreme_value_distribution random number distribution produces random numbers $x$ distributed according to the probability density function

$$p(x \mid a, b) = \frac{1}{b} \exp \left( \frac{a - x}{b} - \exp \left( \frac{a - x}{b} \right) \right).$$

```cpp
template<class RealType = double>
    class extreme_value_distribution {
    public:
        // types
        using result_type = RealType;
        using param_type = unspecified;
        // constructor and reset functions
        explicit extreme_value_distribution(RealType a = 0.0, 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 = 0.0, RealType b = 1.0);
```

2 Requires: $0 < b$.

3 Effects: Constructs an extreme_value_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.

29.6.8.5 Normal distributions

29.6.8.5.1 Class template normal_distribution

A normal_distribution random number distribution produces random numbers $x$ distributed according to the probability density function

$$p(x \mid \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right).$$

The distribution parameters $\mu$ and $\sigma$ are also known as this distribution's mean and standard deviation.

```cpp
template<class RealType = double>
    class normal_distribution {
    public:
        // types
        using result_type = RealType;
        using param_type = unspecified;
```
// constructors and reset functions
explicit normal_distribution(RealType mean = 0.0, 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 = 0.0, 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.

29.6.8.5.2 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

\[
p(x | m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp \left( - \frac{(\ln x - m)^2}{2s^2} \right)
\]

template<class RealType = double>
class lognormal_distribution {
public:
  // types
  using result_type = RealType;
  using param_type = unspecified;

  // constructor and reset functions
  explicit lognormal_distribution(RealType m = 0.0, 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;

§ 29.6.8.5.2


```cpp
result_type max() const;
};

explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);

2 Requires: 0 < s.
3 Effects: Constructs a lognormal_distribution object; m and s correspond to the respective parameters of the distribution.

RealType m() const;
4 Returns: The value of the m parameter with which the object was constructed.

RealType s() const;
5 Returns: The value of the s parameter with which the object was constructed.
```

### 29.6.8.5.3 Class template chi_squared_distribution

A chi_squared_distribution random number distribution produces random numbers \( x > 0 \) distributed according to the probability density function

\[
p(x \mid n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}.
\]

```cpp
template<class RealType = double>
class chi_squared_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    explicit chi_squared_distribution(RealType n = 1);
    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 = 1);

2 Requires: 0 < n.
3 Effects: Constructs a chi_squared_distribution object; n corresponds to the parameter of the distribution.

RealType n() const;
4 Returns: The value of the n parameter with which the object was constructed.
```

### 29.6.8.5.4 Class template cauchy_distribution

A cauchy_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid a, b) = \pi b \left( 1 + \left( \frac{x - a}{b} \right)^2 \right)^{-1}.
\]
template<class RealType = double>
class cauchy_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
    explicit cauchy_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};

explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);

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.

29.6.8.5.5 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

\[
p(x \mid m, n) = \frac{\Gamma((m + n)/2)}{\Gamma(m/2) \Gamma(n/2)} \cdot \left( \frac{m}{n} \right)^{m/2} \cdot x^{(m/2)-1} \cdot \left( 1 + \frac{mx}{n} \right)^{-(m+n)/2}.
\]

template<class RealType = double>
class fisher_f_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

§ 29.6.8.5.5
29.6.8.5.6 Class template student_t_distribution

A student_t_distribution random number distribution produces random numbers \( x \) distributed according to the probability density function

\[
p(x \mid n) = \frac{1}{\sqrt{n\pi}} \cdot \frac{\Gamma((n+1)/2)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}.
\]

```cpp
template<class RealType = double>
class student_t_distribution {
public:
  // types
  using result_type = RealType;
  using param_type = unspecified;

  // constructor and reset functions
  explicit student_t_distribution(RealType n = 1);
  explicit student_t_distribution(const param_type& parm);
  void reset();

  // generating functions
  template<class URBG>
  result_type operator()(URBG& g);
  template<class URBG>
  result_type operator()(URBG& g, const param_type& parm);

  // property functions
  RealType n() const;
  param_type param() const;
  void param(const param_type& parm);
  result_type min() const;
  result_type max() const;
};
```

explicit student_t_distribution(RealType n = 1);

2 Requires: \( 0 < n \).

3 Effects: Constructs a student_t_distribution object; \( n \) corresponds to the parameter of the distribution.

RealType n() const;

4 Returns: The value of the \( n \) parameter with which the object was constructed.
29.6.8.6 Sampling distributions

29.6.8.6.1 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

\[
P(i \mid p_0, \ldots, p_{n-1}) = p_i .
\]

Unless specified otherwise, the distribution parameters are calculated as: \( p_k = w_k / S \) for \( k = 0, \ldots, n-1 \), in which the values \( w_k \), commonly known as the weights, shall be non-negative, non-NaN, and non-infinity. Moreover, the following relation shall hold: \( 0 < S = w_0 + \cdots + w_{n-1} \).

```
template<class IntType = int>
    class discrete_distribution {
    public:
        // types
        using result_type = IntType;
        using param_type = unspecified;

        // constructor and reset functions
        discrete_distribution();
        template<class InputIterator>
            discrete_distribution(InputIterator firstW, InputIterator lastW);
        discrete_distribution(initializer_list<double> wl);
        template<class UnaryOperation>
            discrete_distribution(size_t nw, double xmin, double xmax, UnaryOperation fw);
        explicit discrete_distribution(const param_type& parm);
        void reset();

        // generating functions
        template<class URBG>
            result_type operator()(URBG& g);
        template<class URBG>
            result_type operator()(URBG& g, const param_type& parm);

        // property functions
        vector<double> probabilities() const;
        param_type param() const;
        void param(const param_type& parm);
        result_type min() const;
        result_type max() const;
    };
```

```
discrete_distribution();
```

**Effects:** Constructs a discrete_distribution object with \( n = 1 \) and \( p_0 = 1 \). [Note: Such an object will always deliver the value 0. — end note]

```
template<class InputIterator>
    discrete_distribution(InputIterator firstW, InputIterator lastW);
```

**Requires:** InputIterator shall satisfy the requirements of an input iterator (27.2.3). Moreover, iterator_traits<InputIterator>::value_type shall denote a type that is convertible to double. If firstW == lastW, let \( n = 1 \) and \( w_0 = 1 \). Otherwise, \([\text{firstW}, \text{lastW}]\) 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 (23.14) whose return type shall be convertible to double. Moreover, double shall be convertible to the type of UnaryOperation’s

§ 29.6.8.6.1
sole parameter. If \( nw = 0 \), let \( n = 1 \), otherwise let \( n = nw \). The relation \( 0 < \delta = (xmax - xmin)/n \) shall hold.

**Effects:** Constructs a `discrete_distribution` object with probabilities given by the formula above, using the following values: If \( nw = 0 \), let \( w_0 = 1 \). Otherwise, let \( w_k = fw(xmin + k \cdot \delta + \delta/2) \) for \( k = 0, \ldots, n-1 \).

**Complexity:** The number of invocations of \( fw \) shall not exceed \( n \).

```cpp
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 \( k = 0, \ldots, n-1 \).

### 29.6.8.6.2 Class template `piecewise_constant_distribution` [rand.dist.samp.pconst]

A `piecewise_constant_distribution` random number distribution produces random numbers \( x, b_0 \leq x < b_n \), uniformly distributed over each subinterval \( [b_i, b_{i+1}) \) according to the probability density function

\[
p(x \mid b_0, \ldots, b_n, \rho_0, \ldots, \rho_{n-1}) = \rho_i, \text{ for } b_i \leq x < b_{i+1}.
\]

The \( n + 1 \) distribution parameters \( b_i \), also known as this distribution’s `interval boundaries`, shall satisfy the relation \( b_i < b_{i+1} \) for \( i = 0, \ldots, n-1 \). Unless specified otherwise, the remaining \( n \) distribution parameters are calculated as:

\[
p_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \text{ for } k = 0, \ldots, n-1,
\]

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 RealType = double>
class piecewise_constant_distribution {
public:
    // types
    using result_type = RealType;
    using param_type = unspecified;

    // constructor and reset functions
    piecewise_constant_distribution();
    template<class InputIteratorB, class InputIteratorW>
    piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                     InputIteratorW firstW);
    template<class UnaryOperation>
    piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);
    template<class UnaryOperation>
    piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax,
                                     UnaryOperation fw);
    explicit piecewise_constant_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
    result_type operator()(URBG& g);
    template<class URBG>
    result_type operator()(URBG& g, const param_type& parm);

    // property functions
    vector<result_type> intervals() const;
    vector<result_type> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
};
```
piecewise_constant_distribution();

Effects: Constructs a piecewise_constant_distribution object with \( n = 1 \), \( \rho_0 = 1 \), \( b_0 = 0 \), and \( b_1 = 1 \).

\[ \text{template\langle\text{class InputIteratorB, class InputIteratorW}\rangle} \]
\[ \text{piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,} \]
\[ \text{InputIteratorW firstW);} \]

Requires: InputIteratorB and InputIteratorW shall each satisfy the requirements of an input iterator (Table 95) type. Moreover, 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 \( n = 1 \), \( w_0 = 1 \), \( b_0 = 0 \), 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 \( n \), and any \( w_k \) for \( k \geq n \) shall be ignored by the distribution.

Effects: Constructs a piecewise_constant_distribution object with parameters as specified above.

\[ \text{template\langle\text{class UnaryOperation}\rangle} \]
\[ \text{piecewise_constant_distribution(initializer_list<RealType> bl, UnaryOperation fw);} \]

Requires: Each instance of type UnaryOperation shall be a function object (23.14) 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 \( n = 1 \), \( w_0 = 1 \), \( b_0 = 0 \), and \( b_1 = 1 \). Otherwise, let \([bl.begin(), bl.end()]\) form a sequence \( b_0, \ldots, b_n \), and let \( w_k = fw((b_{k+1} + b_k)/2) \) for \( k = 0, \ldots, n-1 \).

Complexity: The number of invocations of \( fw \) shall not exceed \( n \).

\[ \text{template\langle\text{class UnaryOperation}\rangle} \]
\[ \text{piecewise_constant_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);} \]

Requires: Each instance of type UnaryOperation shall be a function object (23.14) 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_constant_distribution object with parameters taken or calculated from the following values: Let \( b_k = xmin + k \cdot \delta \) for \( k = 0, \ldots, n \), and \( w_k = fw(b_k + \delta/2) \) for \( k = 0, \ldots, n-1 \).

Complexity: The number of invocations of \( fw \) shall not exceed \( n \).

\[ \text{vector\langle result_type \rangle intervals() const;} \]

Returns: A vector\langle result_type \rangle whose size member returns \( n + 1 \) and whose operator[] member returns \( b_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

\[ \text{vector\langle result_type \rangle densities() const;} \]

Returns: A vector\langle result_type \rangle whose size member returns \( n \) and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n - 1 \).

29.6.8.6.3 Class template \textit{piecewise_linear_distribution} [and.dist.samp.plinear]

A piecewise_linear_distribution random number distribution produces random numbers \( x, b_0 \leq x < b_n \), distributed over each subinterval \([b_i, b_{i+1})\) according to the probability density function

\[
p(x \mid b_0, \ldots, b_n, \rho_0, \ldots, \rho_n) = \rho_i \cdot \frac{b_{i+1} - x}{b_{i+1} - b_i} \cdot \frac{x - b_i}{b_{i+1} - b_i}, \quad \text{for } b_i \leq x < b_{i+1}.
\]

The \( n + 1 \) distribution parameters \( b_i \), also known as this distribution’s 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:

\[
0 < S = \frac{1}{2} \sum_{k=0}^{n-1} (w_k + w_{k+1}) \cdot (b_{k+1} - b_k).
\]
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 \( b_1 = 1. \)

Effects: Constructs a piecewise_linear_distribution object with parameters as specified above.

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, \( [bl.begin(), bl.end()] \) form a sequence \( b_0, \ldots, b_n, \) and let \( w_k = f_w(b_k) \) for \( k = 0, \ldots, n. \)

Complexity: The number of invocations of \( f_w \) shall not exceed \( n + 1. \)
template<class UnaryOperation>
piecewise_linear_distribution(size_t nw, RealType xmin, RealType xmax, UnaryOperation fw);

9 Requires: Each instance of type UnaryOperation shall be a function object (23.14) 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 \( 0 < \delta = (x_{\text{max}} - x_{\text{min}})/n \) shall hold.

10 Effects: Constructs a piecewise_linear_distribution object with parameters taken or calculated from the following values: Let \( b_k = x_{\text{min}} + k \cdot \delta \) for \( k = 0, \ldots, n \), and \( w_k = fw(b_k) \) for \( k = 0, \ldots, n \).

11 Complexity: The number of invocations of \( fw \) shall not exceed \( n + 1 \).

vector<result_type> intervals() const;

12 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;

13 Returns: A vector<result_type> whose size member returns \( n \) and whose operator[] member returns \( \rho_k \) when invoked with argument \( k \) for \( k = 0, \ldots, n \).

29.6.9 Low-quality random number generation [c.math.rand]

1 [Note: The header <cstdlib> (21.2.2) declares the functions described in this subclause. —end note]

int rand();
void srand(unsigned int seed);

2 Effects: The rand and srand functions have the semantics specified in the C standard library.

3 Remarks: The implementation may specify that particular library functions may call rand. It is implementation-defined whether the rand function may introduce data races (20.5.5.9). [Note: The other random number generation facilities in this document (29.6) 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

29.7 Numeric arrays [numarray]

29.7.1 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 generalized 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 T&);
template<class T> valarray<T> operator* (const 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 T&);
template<class T> valarray<T> operator/ (const 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 T&);
template<class T> valarray<T> operator%(const T&, const valarray<T>&);

§ 29.7.1
template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const 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 T&);
template<class T> valarray<T> operator- (const 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 T&);
template<class T> valarray<T> operator^ (const 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 T&);
template<class T> valarray<T> operator& (const 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 T&);
template<class T> valarray<T> operator| (const 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 T&);
template<class T> valarray<T> operator<<(const 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 T&);
template<class T> valarray<T> operator>>(const 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 T&);
template<class T> valarray<bool> operator&&(const 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 T&);
template<class T> valarray<bool> operator||(const 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 T&);
template<class T> valarray<bool> operator==(const 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 T&);
template<class T> valarray<bool> operator<(const 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 T&);
template<class T> valarray<bool> operator<=(const 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 T&);
template<class T> valarray<bool> operator>(const 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 T&);
template<class T> valarray<bool> operator>=(const T&, 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 T&);
template<class T> valarray<T> atan2(const T&, 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 T&);
template<class T> valarray<T> pow(const T&, const valarray<T>&);

template<class T> valarray<T> sin (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
template<class T> valarray<T> tan (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);

template<class T> unspecified1 begin(valarray<T>& v);
template<class T> unspecified2 begin(const valarray<T>& v);
template<class T> unspecified1 end(valarray<T>& v);
template<class T> unspecified2 end(const valarray<T>& v);

1 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.

2 The `valarray` array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

3 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.\(^{278}\)

4 Implementations introducing such replacement types shall provide additional functions and operators as follows:

\(^{4.1}\) — for every function taking a `const valarray<T>&` other than `begin` and `end` (29.7.10), identical functions taking the replacement types shall be added;

\(^{4.2}\) — for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

5 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.

6 These library functions are permitted to throw a `bad_alloc` (21.6.3.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

29.7.2 Class template valarray

29.7.2.1 Class template valarray overview

namespace std {
  template<class T> class valarray {
    public:
      using value_type = T;

      // 29.7.2.2, construct/destroy
      valarray();
      explicit valarray(size_t);
      valarray(const T&, size_t);
      valarray(const T*, size_t);
      valarray(const valarray&);
      valarray(valarray&&) noexcept;
      valarray(valarray&&) noexcept;
      valarray(const slice_array<T>&);

278) Annex B recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
valarray(initializer_list<T>);
~valarray();

// 29.7.2.3, assignment
valarray& operator=(const valarray&);
valarray& operator=(valarray&&) noexcept;
valarray& operator=(initializer_list<T>);
valarray& operator=(const T&);
valarray& operator=(const slice_array<T>&);
valarray& operator=(const gslice_array<T>&);
valarray& operator=(const mask_array<T>&);
valarray& operator=(const indirect_array<T>&);

// 29.7.2.4, element access
const T& operator[](size_t) const;
T& operator[](size_t);

// 29.7.2.5, subset operations
valarray operator[](slice) const;
slice_array<T> operator[](slice);
valarray operator[](const gslice&);
gslice_array<T> operator[](const gslice&);
valarray operator[](const valarray<bool>&) const;
mask_array<T> operator[](const valarray<bool>&);
valarray operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);

// 29.7.2.6, unary operators
valarray operator+() const;
valarray operator-() const;
valarray operator~() const;
valarray<bool> operator!() const;

// 29.7.2.7, 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 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&);

// 29.7.2.8, member functions
void swap(valarray&) noexcept;

size_t size() const;
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 29.7. 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.\(^{279}\)

### 29.7.2.2 `valarray` constructors

`valarray();`

*Effects:* Constructs a `valarray` that has zero length.\(^{280}\)

`explicit valarray(size_t n);`

*Effects:* Constructs a `valarray` that has length `n`. Each element of the array is value-initialized (11.6).

`valarray(const T& v, size_t n);`

*Effects:* Constructs a `valarray` that has length `n`. Each element of the array is initialized with `v`.

`valarray(const T* p, size_t n);`

*Requires:* `p` points to an array (11.3.4) 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.\(^{281}\)

`valarray(const valarray& v);`

*Effects:* Constructs a `valarray` that has the same length as `v`. The elements are initialized with the values of the corresponding elements of `v`.\(^{282}\)

`valarray(valarray&& v) noexcept;`

*Effects:* Constructs a `valarray` that has the same length as `v`. The elements are initialized with the values of the corresponding elements of `v`.

*Complexity:* Constant.

`valarray(initializer_list<T> il);`

*Effects:* Equivalent to `valarray(il.begin(), il.size())`.

`valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);`

These conversion constructors convert one of the four reference templates to a `valarray`.\(^{283}\)

---

\(^{279}\) 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.

\(^{280}\) 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.

\(^{281}\) This constructor is the preferred method for converting a C array to a `valarray` object.

\(^{282}\) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they shall implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

\(^{283}\) This indicates that the `valarray` class is a one-dimensional array.
~valarray();

Effects: The destructor is applied to every element of *this; an implementation may return all allocated memory.

29.7.2.3 valarray assignment [valarray.assign]

valarray& operator=(const valarray& v);

Effects: Each element of the *this array is assigned the value of the corresponding element of v. If the length of v is not equal to the length of *this, resizes *this to make the two arrays the same length, as if by calling resize(v.size()), before performing the assignment.

Postconditions: size() == v.size().

Returns: *this.

valarray& operator=(valarray&& v) noexcept;

Effects: *this obtains the value of v. The value of v after the assignment is not specified.

Returns: *this.

Complexity: Linear.

valarray& operator=(initializer_list<T> il);

Effects: Equivalent to: return *this = valarray(il);

valarray& operator=(const T& v);

Effects: Assigns v to each element of *this.

Returns: *this.

valarray& operator=(const slice_array<T>&);
valarray& operator=(const gslice_array<T>&);
valarray& operator=(const mask_array<T>&);
valarray& operator=(const indirect_array<T>&);

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.

29.7.2.4 valarray element access [valarray.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 \&a[i+j] == \&a[i] + j evaluates to true for all size_t i and size_t j such that i+j < a.size(). The expression \&a[i] != \&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) (29.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.
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.

```
const valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
-- end example
```

```
slice_array<T> operator[](slice slicearr);
```

Returns: An object that holds references to elements of the controlled sequence selected by slicearr.

```
valarray<char> v0("abcdefghijklmnop", 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;
```

Returns: A valarray containing those elements of the controlled sequence designated by gslicearr.

```
const valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns
// valarray<char>("dfhkmo", 6)
-- end example
```

```
gslice_array<T> operator[](const gslice& gslicearr);
```

Returns: An object that holds references to elements of the controlled sequence selected by gslicearr.

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDEF", 6);
const size_t lv[] = { 2, 3 };
const size_t dv[] = { 7, 2 };
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBghCijDlEmFp", 16)
-- end example
```

```
valarray operator[](const valarray<bool>& boolarr) const;
```

Returns: A valarray containing those elements of the controlled sequence designated by boolarr.

```
const valarray<char> v0("abcdefghijklmnop", 16);
const bool vb[] = { false, false, true, true, false, true };
// v0[valarray<bool>(vb, 6)] returns
// valarray<char>("cdf", 3)
-- end example
```
mask_array<T> operator[](const valarray<bool>& boolarr);

Returns: An object that holds references to elements of the controlled sequence selected by boolarr.

[Example:]
valarray<char> v0("abcdefgijklmnop", 16);
valarray<char> v1("ABC", 3);
const bool vb[] = { false, false, true, true, false, true };
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)
—end example]

valarray operator[](const valarray<size_t>& indarr) const;

Returns: A valarray containing those elements of the controlled sequence designated by indarr.

[Example:]
const valarray<char> v0("abcdefgijklmnop", 16);
const size_t vi[] = { 7, 5, 2, 3, 8 };
// v0[valarray<size_t>(vi, 5)] returns
// valarray<char>("hfcdi", 5)
—end example]

indirect_array<T> operator[](const valarray<size_t>& indarr);

Returns: An object that holds references to elements of the controlled sequence selected by indarr.

[Example:]
valarray<char> v0("abcdefgijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t vi[] = { 7, 5, 2, 3, 8 };
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
—end example]

29.7.2.6 valarray unary operators

valarray operator+() const;
valarray operator-() const;
valarray operator~() const;
valarray<bool> operator!() const;

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!).

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.

29.7.2.7 valarray 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);

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.
Effects: Each of these operators performs the indicated operation on each of the elements of \( \ast\text{this} \) and the corresponding element of \( v \).

Returns: \( \ast\text{this} \).

Remarks: The appearance of an array on the left-hand side of a compound assignment does not invalidate references or pointers.

```cpp
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 \( \ast\text{this} \) and \( v \).

Returns: \( \ast\text{this} \)

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.

### 29.7.2.8 valarray member functions

#### void swap(valarray& v) noexcept;

Effects: \( \ast\text{this} \) obtains the value of \( v \). \( v \) obtains the value of \( \ast\text{this} \).

Complexity: Constant.

size_t size() const;

Returns: The number of elements in the array.

Complexity: Constant time.

\( T \) sum() const;

Requires: \( \text{size()} > 0 \). This function may only be instantiated for a type \( T \) to which \( \text{operator+=} \) can be applied.

Returns: The sum of all the elements of the array. If the array has length 1, returns the value of element 0. Otherwise, the returned value is calculated by applying \( \text{operator+=} \) to a copy of an element of the array and all other elements of the array in an unspecified order.

\( T \) min() const;

Requires: \( \text{size()} > 0 \)

Returns: The minimum value contained in \( \ast\text{this} \). For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using \( \text{operator<} \).

\( T \) max() const;

Requires: \( \text{size()} > 0 \).

Returns: The maximum value contained in \( \ast\text{this} \). For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using \( \text{operator<} \).

valarray shift(int n) const;

Returns: A valarray of length \( \text{size()} \), each of whose elements \( I \) is \( \ast\text{this}[[I + n]] \) if \( I + n \) is non-negative and less than \( \text{size()} \), 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.

29.7.3 valarray non-member operations

29.7.3.1 valarray binary operators

template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const 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.

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.
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.

29.7.3.2 valarray logical operators

Returns: A valarray<bool> whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

² 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.

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.

4 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 argument and the non-array argument.

29.7.3.3 valarray transcendentals

Returns: A valarray<T> 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.
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>&);

1 Requiring: 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 \( T \).

29.7.3.4 valarray specialized algorithms

```
template<class T> void swap(valarray<T>& x, valarray<T>& y) noexcept;
```

1 Effects: Equivalent to \( x\.swap(y) \).

29.7.4 Class slice

29.7.4.1 Class slice overview

```
namespace std {
    class slice {
        public:
            slice();
            slice(size_t, size_t, size_t);
            size_t start() const;
            size_t size() const;
            size_t stride() const;
    }
}
```

1 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.\(^{283}\)

29.7.4.2 slice constructors

```
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

1 The default constructor is equivalent to \( \text{slice}(0, 0, 0) \). 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.

2 [Example: slice\((3, 8, 2)\) constructs a slice which selects elements 3, 5, 7, ... 17 from an array. —end example]

29.7.4.3 slice access functions

```
size_t start() const;
size_t size() const;
size_t stride() const;
```

1 Returns: The start, length, or stride specified by a slice object.

2 Complexity: Constant time.

29.7.5 Class template slice_array

29.7.5.1 Class template slice_array overview

```
namespace std {
    template<class T> class slice_array {
        public:
            using value_type = T;
    }
}
```

283) 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.
The `slice_array` 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

### 29.7.5.2 slice_array assignment

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.

### 29.7.5.3 slice_array compound assignment

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.

### 29.7.5.4 slice_array fill function

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.

### 29.7.6 The gslice class

**namespace std {**

```
class gslice {
public:
    gslice();
};
```
gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

size_t start() const;
valarray<size_t> size() const;
valarray<size_t> stride() const;

1 This class represents a generalized slice out of an array. A gslice is defined by a starting offset (s), a set of lengths (lj), and a set of strides (dj). The number of lengths shall equal the number of strides.

A gslice represents a mapping from a set of indices (ij), 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

\[
\begin{align*}
\text{start} & = 3 \\
\text{length} & = \{2, 4, 3\} \\
\text{stride} & = \{19, 4, 1\}
\end{align*}
\]

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:

\[
\begin{array}{cccc}
(i_0, & i_1, & i_2, & k) = \\
(0, 0, 0, 3), & (0, 0, 1, 4), & (0, 0, 2, 5), & (0, 1, 0, 7), \\
(0, 1, 1, 8), & (0, 1, 2, 9), & (0, 2, 0, 11), & (0, 2, 1, 12), \\
(0, 2, 2, 13), & (0, 3, 0, 15), & (0, 3, 1, 16), & (0, 3, 2, 17), \\
(1, 0, 0, 22), & (1, 0, 1, 23), & \ldots \end{array}
\]

That is, the highest-ordered index turns fastest. —end example]

2 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

\[
\begin{array}{cccc}
(0, 0, 0, 3), & (0, 0, 1, 4), & (0, 0, 2, 5), & (0, 1, 0, 4), \\
(0, 1, 1, 5), & (0, 1, 2, 6), & \ldots \end{array}
\]
If a degenerate slice is used as the argument to the non-\texttt{const} version of \texttt{operator[]} (\texttt{const gslice&}), the behavior is undefined.

### 29.7.6.2 \texttt{gslice} constructors

\begin{verbatim}
gslice();
gslice(size_t start, const valarray&lt;size_t&gt;& lengths,
       const valarray&lt;size_t&gt;& strides);
gslice(const gslice&);
\end{verbatim}

The default constructor is equivalent to \texttt{gslice(0, valarray&lt;size_t&gt;(), valarray&lt;size_t&gt;())}. The constructor with arguments builds a \texttt{gslice} based on a specification of start, lengths, and strides, as explained in the previous subclause.

### 29.7.6.3 \texttt{gslice} access functions

\begin{verbatim}
size_t start() const;
valarray&lt;size_t&gt; size() const;
valarray&lt;size_t&gt; stride() const;
\end{verbatim}

\textbf{Returns:} The representation of the start, lengths, or strides specified for the \texttt{gslice}.

\textbf{Complexity:} \texttt{start()} is constant time. \texttt{size()} and \texttt{stride()} are linear in the number of strides.

### 29.7.7 Class template \texttt{gslice_array}

#### 29.7.7.1 Class template \texttt{gslice_array} overview

\begin{verbatim}
namespace std {
  template<class T> class gslice_array {
    public:
      using value_type = T;

      void operator= (const valarray&lt;T&gt;&) const;
      void operator*= (const valarray&lt;T&gt;&) const;
      void operator/= (const valarray&lt;T&gt;&) const;
      void operator%= (const valarray&lt;T&gt;&) const;
      void operator+= (const valarray&lt;T&gt;&) const;
      void operator-= (const valarray&lt;T&gt;&) const;
      void operator^= (const valarray&lt;T&gt;&) const;
      void operator&= (const valarray&lt;T&gt;&) const;
      void operator|= (const valarray&lt;T&gt;&) const;
      void operator&=(const valarray&lt;T&gt;&) const;
      void operator>>=(const valarray&lt;T&gt;&) const;

      gslice_array(const gslice_array&);
      gslice_array() = delete;       // as implied by declaring copy constructor above
  }
}
\end{verbatim}

This template is a helper template used by the \texttt{slice} subscript operator

\begin{verbatim}
gslice_array&lt;T&gt; valarray&lt;T&gt;::operator[](const gslice&);
\end{verbatim}

It has reference semantics to a subset of an array specified by a \texttt{gslice} object.

Thus, the expression \texttt{a[gslice(1, length, stride)] = b} has the effect of assigning the elements of \texttt{b} to a generalized slice of the elements in \texttt{a}.

### 29.7.7.2 \texttt{gslice_array} assignment

\begin{verbatim}
void operator=(const valarray&lt;T&gt;&) const;
\end{verbatim}
const gslice_array& operator=(const gslice_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 gslice_array refers.

### 29.7.7.3 gslice_array compound assignment

```cpp
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;
```

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.

### 29.7.7.4 gslice_array fill function

```cpp
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 gslice_array object refers.

### 29.7.8 Class template mask_array

#### 29.7.8.1 Class template mask_array overview

```cpp
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;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    mask_array(const mask_array&);
    ~mask_array();
    const mask_array& operator=(const mask_array&) const;
    void operator=(const T&) const;

    mask_array() = delete; // as implied by declaring copy constructor above
  }
}
```

This template is a helper template used by the mask subscript operator:

```cpp
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.)
29.7.8.2 mask_array assignment

```cpp
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

29.7.8.3 mask_array compound assignment

```cpp
void operator*=(const valarray<T>&) const;
void operator/=(const valarray<T>&) const;
void operator%=(const valarray<T>&) const;
void operator+=(const valarray<T>&) const;
void operator-=(const valarray<T>&) const;
void operator^=(const valarray<T>&) const;
void operator&=(const valarray<T>&) const;
void operator|=(const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

29.7.8.4 mask_array fill function

```cpp
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

29.7.9 Class template indirect_array

29.7.9.1 Class template indirect_array overview

```cpp
namespace std {
    template<class T> class indirect_array {
        using value_type = T;
        void operator=(const valarray<T>&) const;
        void operator*=(const valarray<T>&) const;
        void operator/=(const valarray<T>&) const;
        void operator%=(const valarray<T>&) const;
        void operator+=(const valarray<T>&) const;
        void operator-=(const valarray<T>&) const;
        void operator^=(const valarray<T>&) const;
        void operator&=(const valarray<T>&) const;
        void operator|=(const valarray<T>&) const;
        void operator<<=(const valarray<T>&) const;
        void operator>>=(const valarray<T>&) const;
        indirect_array(const indirect_array&);
        ~indirect_array();
        const indirect_array& operator=(const indirect_array&); // as implied by declaring copy constructor above
        void operator=(const T&);
        indirect_array() = delete; // as implied by declaring copy constructor above
    };
}
```

1 This template is a helper template used by the indirect subscript operator

```cpp
indirect_array<T> valarray<T>::operator[](const valarray<size_t>&);
```

2 It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression `a[indirect] = b;` has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`. 
29.7.9.2 indirect_array assignment

void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;

These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the valarray<T> object to which it refers.

If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

[Example:
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]

29.7.9.3 indirect_array compound assignment

void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+=(const valarray<T>&) const;
void operator-=(const valarray<T>&) const;
void operator^=(const valarray<T>&) const;
void operator&=(const valarray<T>&) const;
void operator|=(const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;

These compound assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the valarray<T> object to which the indirect_array object refers.

If the indirect_array specifies an element in the valarray<T> object to which it refers more than once, the behavior is undefined.

29.7.9.4 indirect_array fill function

void operator=(const T&) const;

This function has reference semantics, assigning the value of its argument to the elements of the valarray<T> object to which the indirect_array object refers.

29.7.10 valarray range access

In the begin and end function templates that follow, unspecified1 is a type that meets the requirements of a mutable random access iterator (27.2.7) and of a contiguous iterator (27.2.1) whose value_type is the template parameter T and whose reference type is T&. unspecified2 is a type that meets the requirements of a constant random access iterator (27.2.7) and of a contiguous iterator (27.2.1) whose value_type is the template parameter T and whose reference type is const T&.

The iterators returned by begin and end for an array are guaranteed to be valid until the member function resize(size_t, T) (29.7.2.8) is called for that array or until the lifetime of that array ends, whichever happens first.

template<class T> unspecified1 begin(valarray<T>& v);
template<class T> unspecified2 begin(const valarray<T>& v);

Returns: An iterator referencing the first value in the array.

template<class T> unspecified1 end(valarray<T>& v);
template<class T> unspecified2 end(const valarray<T>& v);

Returns: An iterator referencing one past the last value in the array.
namespace std {
    // 29.8.2, 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);

    // 29.8.3, 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, ForwardIterator first, ForwardIterator last);
    template<class ExecutionPolicy, class ForwardIterator, class T>
    T reduce(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, T init);
    template<class ExecutionPolicy, class ForwardIterator, class T, class BinaryOperation>
    T reduce(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, T init, BinaryOperation binary_op);

    // 29.8.4, 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);

    // 29.8.5, 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, 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 28.4.5  
                   ForwardIterator1 first1, ForwardIterator1 last1,  
                   ForwardIterator2 first2,  
                   T init,  
                   BinaryOperation1 binary_op1,  
                   BinaryOperation2 binary_op2);

template<class ExecutionPolicy,  
     class ForwardIterator, class T,  
     class BinaryOperation, class UnaryOperation>
T transform_reduce(ExecutionPolicy&& exec, // see 28.4.5  
                   ForwardIterator first, ForwardIterator last,  
                   T init,  
                   BinaryOperation binary_op, UnaryOperation unary_op);

// 29.8.6, partial sum
template<class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first,  
                            InputIterator last,  
                            OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first,  
                            InputIterator last,  
                            OutputIterator result,  
                            BinaryOperation binary_op);

// 29.8.7, exclusive scan
template<class InputIterator, class OutputIterator, class T>
OutputIterator exclusive_scan(InputIterator first, InputIterator last,  
                               OutputIterator result,  
                               T init);

template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
OutputIterator exclusive_scan(InputIterator first, InputIterator last,  
                               OutputIterator result,  
                               T init, BinaryOperation binary_op);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 28.4.5  
                                ForwardIterator1 first, ForwardIterator1 last,  
                                ForwardIterator2 result,  
                                T init);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation>
ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec, // see 28.4.5  
                                ForwardIterator1 first, ForwardIterator1 last,  
                                ForwardIterator2 result,  
                                T init, BinaryOperation binary_op);

// 29.8.8, inclusive scan
template<class InputIterator, class OutputIterator>
OutputIterator inclusive_scan(InputIterator first, InputIterator last,  
                                 OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator inclusive_scan(InputIterator first, InputIterator last,  
                                 OutputIterator 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>
ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5  
                                ForwardIterator1 first, ForwardIterator1 last,  
                                ForwardIterator2 result,  
                                T init);
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class BinaryOperation>
ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec, // see 28.4.5
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 28.4.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
BinaryOperation binary_op, T init);

// 29.8.9, 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 28.4.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
T init,
BinaryOperation binary_op,
UnaryOperation unary_op);

// 29.8.10, 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 28.4.5
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 28.4.5
ForwardIterator1 first, ForwardIterator1 last,
ForwardIterator2 result,
BinaryOperation binary_op,
UnaryOperation unary_op,
T init);
// 29.8.11, 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,
                                        ForwardIterator1 first,
                                        ForwardIterator1 last,
                                        ForwardIterator2 result);

template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation>
  ForwardIterator2 adjacent_difference(ExecutionPolicy&& exec,
                                        ForwardIterator1 first,
                                        ForwardIterator1 last,
                                        ForwardIterator2 result,
                                        BinaryOperation binary_op);

// 29.8.12, iota
template<class ForwardIterator, class T>
  void iota(ForwardIterator first, ForwardIterator last, T value);

// 29.8.13, greatest common divisor
template<class M, class N>
  constexpr common_type_t<M,N> gcd(M m, N n);

// 29.8.14, least common multiple
template<class M, class N>
  constexpr common_type_t<M,N> lcm(M m, N n);

1 The requirements on the types of algorithms’ arguments that are described in the introduction to Clause 28 also apply to the following algorithms.

2 Throughout this subclause, the parameters UnaryOperation, BinaryOperation, BinaryOperation1, and BinaryOperation2 are used whenever an algorithm expects a function object (23.14).

[Note: The use of closed ranges as well as semi-open ranges to specify requirements throughout this subclause is intentional. —end note]

29.8.2 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 meet the requirements of CopyConstructible (Table 24) and CopyAssignable (Table 26) types. In the range [first, last], binary_op shall neither modify elements nor invalidate iterators or subranges.284

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.285

284) The use of fully closed ranges is intentional.

285) 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.
29.8.3 Reduce

```
template<class InputIterator>
    typename iterator_traits<InputIterator>::value_type
    reduce(InputIterator first, InputIterator last);
```

1  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);
```

2  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);
```

3  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);
```

4  Effects: Equivalent to:
    return reduce(std::forward<ExecutionPolicy>(exec), first, last, init, plus<>());

```
template<class InputIterator, class T, class BinaryOperation>
    T reduce(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

5  Requires:
    (5.1)  T shall be MoveConstructible (Table 23).
    (5.2)  All of binary_op(init, *first), binary_op(*first, init), binary_op(init, init), and
            binary_op(*first, *first) shall be convertible to T.
    (5.3)  binary_op shall neither invalidate iterators or subranges, nor modify elements in the range
            [first, last].

6  Returns: GENERALIZED_SUM(binary_op, init, *i, ...) for every i in [first, last).

7  Complexity: \( \Theta(last - first) \) applications of binary_op.

8  [Note: The difference between reduce and accumulate is that reduce applies binary_op in an
    unspecified order, which yields a nondeterministic result for non-associative or non-commutative
    binary_op such as floating-point addition. — end note]

29.8.4 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,
```

§ 29.8.4
BinaryOperation2 binary_op2);

1 **Requires:** T shall meet the requirements of CopyConstructible (Table 24) and CopyAssignable (Table 26) types. In the ranges \([\text{first1}, \text{last1}]\) and \([\text{first2}, \text{first2} + (\text{last1} - \text{first1})]\) binary_op1 and binary_op2 shall neither modify elements nor invalidate iterators or subranges.286

2 **Effects:** Computes its result by initializing the accumulator acc with the initial value init and then modifying it with \(\text{acc} = \text{std}::\text{move}(\text{acc}) + (*\text{i1}) \ast (*\text{i2})\) or \(\text{acc} = \text{binary_op1}(\text{std}::\text{move}(\text{acc}), \text{binary_op2}(*\text{i1}, *\text{i2}))\) for every iterator i1 in the range \([\text{first1}, \text{last1}]\) and iterator i2 in the range \([\text{first2}, \text{first2} + (\text{last1} - \text{first1})]\) in order.

### 29.8.5 Transform reduce

**template<**class InputIterator1, class InputIterator2, class T>
T transform_reduce(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, T init);

**template<**class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
T transform_reduce(ExecutionPolicy&& exec, ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, T init);

1 **Effects:** Equivalent to:

\[
\text{return transform_reduce(first1, last1, first2, init, plus<>(), multiplies<>());}\]

**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 ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T, class BinaryOperation1, class BinaryOperation2>
T transform_reduce(ExecutionPolicy&& exec, ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, T init, BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

2 **Requires:**

1. T shall be MoveConstructible (Table 23).
2. All of
3. — binary_op1(init, init),
4. — binary_op1(init, binary_op2(*first1, *first2)),
5. — binary_op1(binary_op2(*first1, *first2), init), and
6. — binary_op1(binary_op2(*first1, *first2), binary_op2(*first1, *first2)) shall be convertible to T.
7. — Neither binary_op1 nor binary_op2 shall invalidate subranges, or modify elements in the ranges \([\text{first1}, \text{last1}]\) and \([\text{first2}, \text{first2} + (\text{last1} - \text{first1})]\).

3 **Returns:**

\[
\text{GENERALIZED\_SUM(binary_op1, init, binary_op2(*i, *(first2 + (i - first1))), ...)}
\]
for every iterator i in \([\text{first1}, \text{last1}]\).

4 **Complexity:** \(\Theta(\text{last1} - \text{first1})\) applications each of binary_op1 and binary_op2.

286) The use of fully closed ranges is intentional.
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 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:
(5.1) — T shall be MoveConstructible (Table 23).
(5.2) — All of
  (5.2.1) — binary_op(init, init),
  (5.2.2) — binary_op(init, unary_op(*first)),
  (5.2.3) — binary_op(unary_op(*first), init), and
  (5.2.4) — binary_op(unary_op(*first), unary_op(*first))
  shall be convertible to T.
(5.3) — Neither unary_op nor binary_op shall invalidate subranges, or modify elements in the range [first, last].

Returns:
GENERALIZED_SUM(binary_op, init, unary_op(*i), ...)
for every iterator i in [first, last).

Complexity: \(O(last - first)\) applications each of unary_op and binary_op.

[Note: transform_reduce does not apply unary_op to init. — end note]

29.8.6 Partial sum

template<class InputIterator, class OutputIterator>
OutputIterator partial_sum(  
  InputIterator first, InputIterator last,  
  OutputIterator result);

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 (27.2.1) to the result output iterator. In the ranges [first, last] and [result, result + (last - first)] binary_op shall neither modify elements nor invalidate iterators or subranges.  

Effects: For a non-empty range, the function creates an accumulator acc whose type is InputIterator’s value type, initializes it with *first, and assigns the result to *result. For every iterator i in [first + 1, last) in order, acc is then modified by acc = std::move(acc) + *i or acc = binary_op(std::move(acc), *i) and the result is assigned to *(result + (i - first)).

Returns: result + (last - first).

Complexity: Exactly (last - first) - 1 applications of the binary operation.

Remarks: result may be equal to first.

---

\[287\] The use of fully closed ranges is intentional.
29.8.7 Exclusive scan

```cpp
template<class InputIterator, class OutputIterator, class T>
OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                               OutputIterator result, T init);
```

1. **Effects:** Equivalent to:
   ```cpp
   return exclusive_scan(first, last, result, init, plus<>());
   ```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2, class T>
ForwardIterator2 exclusive_scan(ExecutionPolicy&& exec,
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result, T init);
```

2. **Effects:** Equivalent to:
   ```cpp
   return exclusive_scan(std::forward<ExecutionPolicy>(exec),
                          first, last, result, init, plus<>());
   ```

```cpp
template<class InputIterator, class OutputIterator, class T, class BinaryOperation>
OutputIterator exclusive_scan(InputIterator first, InputIterator last,
                               OutputIterator result, T init, BinaryOperation binary_op);
```

3. **Requires:**
   - T shall be MoveConstructible (Table 23).
   - All of binary_op(init, init), binary_op(init, *first), and binary_op(*first, *first) shall be convertible to T.
   - binary_op shall neither invalidate iterators or subranges, nor modify elements in the ranges [first, last] or [result, result + (last - first)].

4. **Effects:** For each integer K in [0, last - first) assigns through result + K the value of:
   ```cpp
   GENERALIZED_NONCOMMUTATIVE_SUM(
       binary_op, init, *(first + 0), *(first + 1), ..., *(first + K - 1))
   ```

5. **Returns:** The end of the resulting range beginning at result.

6. **Complexity:** $O(last - first)$ applications of binary_op.

7. **Remarks:** result may be equal to first.

8. **[Note:** The difference between exclusive_scan and inclusive_scan is that exclusive_scan excludes the i-th input element from the i-th sum. If binary_op is not mathematically associative, the behavior of exclusive_scan may be nondeterministic. — end note]

29.8.8 Inclusive scan

```cpp
template<class InputIterator, class OutputIterator>
OutputIterator inclusive_scan(InputIterator first, InputIterator last, OutputIterator result);
```

1. **Effects:** Equivalent to:
   ```cpp
   return inclusive_scan(first, last, result, plus<>());
   ```

```cpp
template<class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 inclusive_scan(ExecutionPolicy&& exec,
                                ForwardIterator1 first, ForwardIterator1 last,
                                ForwardIterator2 result);
```

2. **Effects:** Equivalent to:
   ```cpp
   return inclusive_scan(std::forward<ExecutionPolicy>(exec), first, last, result, plus<>());
   ```

```cpp
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:**

(3.1) If init is provided, T shall be MoveConstructible (Table 23); otherwise, ForwardIterator1’s value type shall be MoveConstructible.

(3.2) 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.

(3.3) 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

(4.1) GENERALIZED_NONCOMMUTATIVE_SUM
    - binary_op, init, *(first + 0), *(first + 1), ..., *(first + K)
    if init is provided, or

(4.2) 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]

### 29.8.9 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 BinaryOperation, class UnaryOperation>
ForwardIterator2 transform_exclusive_scan(ExecutionPolicy&& exec,
    ForwardIterator1 first, ForwardIterator1 last,
    ForwardIterator2 result, T init, BinaryOperation binary_op, UnaryOperation unary_op);

**Requires:**

(1.1) T shall be MoveConstructible (Table 23).

(1.2) All of

(1.2.1) binary_op(init, init),

(1.2.2) binary_op(init, unary_op(*first)), and
binary_op(unary_op(*first), unary_op(*first)) shall be convertible to T.

Neither unary_op nor binary_op shall invalidate iterators or subranges, or 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:

```
GENERALIZED_NONCOMMUTATIVE_SUM
binary_op, init,
unary_op(*(first + 0)), unary_op(*(first + 1)), ..., unary_op(*(first + K - 1))
```

**Returns:** The end of the resulting range beginning at result.

**Complexity:** \( O(last - first) \) applications each of unary_op and binary_op.

**Remarks:** result may be equal to first.

[Note: 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]

### 29.8.10 Transform inclusive scan

```cpp
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 BinaryOperation, class UnaryOperation>
ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec,
                                        ForwardIterator1 first, ForwardIterator1 last,
                                        ForwardIterator2 result,
                                        BinaryOperation binary_op, UnaryOperation unary_op);

template<class InputIterator, class OutputIterator,
         class BinaryOperation, class UnaryOperation, class T>
OutputIterator transform_inclusive_scan(InputIterator first, InputIterator last,
                                        OutputIterator result,
                                        BinaryOperation binary_op, UnaryOperation unary_op,
                                        T init);

template<class ExecutionPolicy,
         class ForwardIterator1, class ForwardIterator2,
         class BinaryOperation, class UnaryOperation, class T>
ForwardIterator2 transform_inclusive_scan(ExecutionPolicy&& exec,
                                        ForwardIterator1 first, ForwardIterator1 last,
                                        ForwardIterator2 result,
                                        BinaryOperation binary_op, UnaryOperation unary_op,
                                        T init);
```

**Requires:**

1. If init is provided, \( T \) shall be MoveConstructible (Table 23); otherwise, ForwardIterator1’s value type shall be MoveConstructible.

2. If init is provided, all of

   - binary_op(init, init),
   - binary_op(init, unary_op(*first)), and
   - 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.

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 \([0, \text{last} - \text{first})\) assigns through \( \text{result} + K \) the value of

\[
\text{— GENERALIZED_NONCOMMUTATIVE_SUM}(
\text{binary_op, init},
\text{unary_op}(*\text{first} + 0), \text{unary_op}(*\text{first} + 1), \ldots, \text{unary_op}(*\text{first} + K))
\]

if init is provided, or

\[
\text{— GENERALIZED_NONCOMMUTATIVE_SUM}(
\text{binary_op},
\text{unary_op}(*\text{first} + 0), \text{unary_op}(*\text{first} + 1), \ldots, \text{unary_op}(*\text{first} + K))
\]

otherwise.

Returns: The end of the resulting range beginning at \( \text{result} \).

Complexity: \( O(\text{last} - \text{first}) \) applications each of \( \text{unary_op} \) and \( \text{binary_op} \).

Remarks: result may be equal to \( \text{first} \).

[Note: The difference between \text{transform_exclusive_scan} and \text{transform_inclusive_scan} is that \text{transform_inclusive_scan} includes the \( i \)th input element in the \( i \)th sum. If \( \text{binary_op} \) is not mathematically associative, the behavior of \text{transform_inclusive_scan} may be nondeterministic. \text{transform_inclusive_scan} does not apply \( \text{unary_op} \) to init. — end note]

29.8.11 Adjacent difference

[adjacent.difference]

\[
\text{template<}\text{class InputIterator, class OutputIterator>}
\]
\[
\text{OutputIterator}
\]
\[
\text{adjacent_difference(InputIterator first, InputIterator last, OutputIterator result);}
\]
\[
\text{template<}\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2>}
\]
\[
\text{ForwardIterator2}
\]
\[
\text{adjacent_difference(ExecutionPolicy\&\& exec,}
\]
\[
\text{ForwardIterator1 first, ForwardIterator1 last, ForwardIterator2 result);}
\]
\[
\text{template<}\text{class InputIterator, class OutputIterator, class BinaryOperation>}
\]
\[
\text{OutputIterator}
\]
\[
\text{adjacent_difference(InputIterator first, InputIterator last,}
\]
\[
\text{OutputIterator result, BinaryOperation binary_op);}
\]
\[
\text{template<}\text{class ExecutionPolicy, class ForwardIterator1, class ForwardIterator2,}
\]
\[
\text{class BinaryOperation>
\]
\[
\text{ForwardIterator2}
\]
\[
\text{adjacent_difference(ExecutionPolicy\&\& exec,}
\]
\[
\text{ForwardIterator1 first, ForwardIterator1 last,}
\]
\[
\text{ForwardIterator2 result, BinaryOperation binary_op);}
\]

Requires:

(1.1) For the overloads with no \text{ExecutionPolicy}, \text{InputIterator}’s value type shall be \text{MoveAssignable} (Table 25) and shall be constructible from the type of \(*\text{first}\. \text{acc} \) (defined below) shall be writable (27.2.1) to the \text{result} output iterator. The result of the expression \( \text{val} - \text{std::move(acc)} \) or \( \text{binary_op(val, std::move(acc))} \) shall be writable to the \text{result} output iterator.

(1.2) For the overloads with an \text{ExecutionPolicy}, the value type of \text{ForwardIterator1} shall be \text{CopyConstructible} (Table 24), constructible from the expression \(*\text{first} - \text{first}\) or \text{binary_op}(*\text{first}, *\text{first}), and assignable to the value type of \text{ForwardIterator2}.

(1.3) 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.\(^{288}\)

Effects: For the overloads with no \text{ExecutionPolicy} and 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, creates an object \text{val} whose type is \text{InputIterator}’s value type, initializes it with \(*i\), computes \( \text{val} - \text{std::move(acc)} \) or \( \text{binary_op(val, std::move(acc))} \), assigns the result to \(*(\text{result} + (i - \text{first}))\), and move assigns from \text{val} to \text{acc}.

For the overloads with an \text{ExecutionPolicy} and a non-empty range, first the function creates an object whose type is \text{ForwardIterator1}’s value type, initializes it with \(*\text{first}\), and assigns the

\(^{288}\) The use of fully closed ranges is intentional.
result to *result. Then for every d in [1, last - first - 1], creates an object val whose type is ForwardIterator’s value type, initializes it with *(first + d) - *(first + d - 1) or binary_op(*(first + d), *(first + d - 1)), and assigns the result to *(result + d).

Returns: result + (last - first).

Complexity: Exactly (last - first) - 1 applications of the binary operation.

Remarks: For the overloads with no ExecutionPolicy, result may be equal to first. For the overloads with an ExecutionPolicy, the ranges [first, last) and [result, result + (last - first)) shall not overlap.

29.8.12 Iota

```
template<class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

Requires: T shall be convertible to ForwardIterator’s value type. The expression ++val, where val has type T, shall be well-formed.

Effects: For each element referred to by the iterator i in the range [first, last), assigns *i = value and increments value as if by ++value.

Complexity: Exactly last - first increments and assignments.

29.8.13 Greatest common divisor

```
template<class M, class N>
constexpr common_type_t<M,N> gcd(M m, N n);
```

Requires: |m| and |n| shall be representable as a value of common_type_t<M, N>. [Note: These requirements ensure, for example, that gcd(m, m) = |m| is representable as a value of type M. —end note]

Remarks: If either M or N is not an integer type, or if either is cv bool, the program is ill-formed.

Returns: Zero when m and n are both zero. Otherwise, returns the greatest common divisor of |m| and |n|.

Throws: Nothing.

29.8.14 Least common multiple

```
template<class M, class N>
constexpr common_type_t<M,N> lcm(M m, N n);
```

Requires: |m| and |n| shall be representable as a value of common_type_t<M, N>. The least common multiple of |m| and |n| shall be representable as a value of common_type_t<M,N>.

Remarks: If either M or N is not an integer type, or if either is cv bool the program is ill-formed.

Returns: Zero when either m or n is zero. Otherwise, returns the least common multiple of |m| and |n|.

Throws: Nothing.

29.9 Mathematical functions for floating-point types

29.9.1 Header <cmath> synopsis

```
namespace std {
    using float_t = see below;
    using double_t = see below;
}
```

#define HUGE_VAL see below
#define HUGE_VALF see below
#define HUGE_VALL see below
#define INFINITY see below
#define NAN see below
#define FP_INFINITE see below
#define FP_NAN see below
#define FP_NORMAL see below
```c
#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 20.2
    double acos(double x);
    long double acos(long double x); // see 20.2
    float acosf(float x);
    long double acosl(long double x);

    float asin(float x); // see 20.2
    double asin(double x);
    long double asin(long double x); // see 20.2
    float asinf(float x);
    long double asinl(long double x);

    float atan(float x); // see 20.2
    double atan(double x);
    long double atan(long double x); // see 20.2
    float atanf(float x);
    long double atanl(long double x);

    float atan2(float y, float x); // see 20.2
    double atan2(double y, double x);
    long double atan2(long double y, long double x); // see 20.2
    float atan2f(float y, float x);
    long double atan2l(long double y, long double x);

    float cos(float x); // see 20.2
    double cos(double x);
    long double cos(long double x); // see 20.2
    float cosf(float x);
    long double cosl(long double x);

    float sin(float x); // see 20.2
    double sin(double x);
    long double sin(long double x); // see 20.2
    float sinf(float x);
    long double sinl(long double x);

    float tan(float x); // see 20.2
    double tan(double x);
    long double tan(long double x); // see 20.2
    float tanf(float x);
    long double tanl(long double x);

    float acosh(float x); // see 20.2
    double acosh(double x);
    long double acosh(long double x); // see 20.2
    float acoshf(float x);
    long double acoshl(long double x);

    float asinh(float x); // see 20.2
    double asinh(double x);
    long double asinh(long double x); // see 20.2
}```
float asinhf(float x);
long double asinhl(long double x);

float atanh(float x);  // see 20.2
double atanh(double x);
long double atanh(long double x);  // see 20.2
float atanhf(float x);
long double atanhl(long double x);

float coshf(float x);  // see 20.2
double cosh(double x);
long double cosh(long double x);  // see 20.2
float coshf(float x);
long double coshl(long double x);

float sinh(float x);  // see 20.2
double sinh(double x);
long double sinh(long double x);  // see 20.2
float sinhf(float x);
long double sinhl(long double x);

float tanhf(float x);  // see 20.2
double tanh(double x);
long double tanh(long double x);  // see 20.2
float tanhf(float x);
long double tanhl(long double x);

float expf(float x);  // see 20.2
double exp(double x);
long double exp(long double x);  // see 20.2
float expf(float x);
long double expl(long double x);

float exp2f(float x);  // see 20.2
double exp2(double x);
long double exp2(long double x);  // see 20.2
float exp2f(float x);
long double exp2l(long double x);

float expm1f(float x);  // see 20.2
double expm1(double x);
long double expm1(long double x);  // see 20.2
float expm1f(float x);
long double expm1l(long double x);

float frexp(float value, int* exp);  // see 20.2
double frexp(double value, int* exp);
long double frexp(long double value, int* exp);  // see 20.2
float frexpf(float value, int* exp);
long double frexpl(long double value, int* exp);

int ilogbf(float x);  // see 20.2
int ilogb(double x);
int ilogb(long double x);  // see 20.2
int ilogbf(float x);
int ilogbl(long double x);

float ldexp(float x, int exp);  // see 20.2
double ldexp(double x, int exp);
long double ldexp(long double x, int exp);  // see 20.2
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
float log(float x); // see 20.2
double log(double x);
long double log(long double x); // see 20.2
float logf(float x);
long double logl(long double x);

float log10(float x); // see 20.2
double log10(double x);
long double log10(long double x); // see 20.2
float log10f(float x);
long double log10l(long double x);

float log1p(float x); // see 20.2
double log1p(double x);
long double log1p(long double x); // see 20.2
float log1pf(float x);
long double log1pl(long double x);

float log2(float x); // see 20.2
double log2(double x);
long double log2(long double x); // see 20.2
float log2f(float x);
long double log2l(long double x);

float logb(float x); // see 20.2
double logb(double x);
long double logb(long double x); // see 20.2
float logbf(float x);
long double logbl(long double x);

float modf(float value, float* iptr); // see 20.2
double modf(double value, double* iptr);
long double modf(long double value, long double* iptr); // see 20.2
float modff(float value, float* iptr);
long double modfl(long double value, long double* iptr);

float scalbn(float x, int n); // see 20.2
double scalbn(double x, int n);
long double scalbn(long double x, int n); // see 20.2
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);

float scalbln(float x, long int n); // see 20.2
double scalbln(double x, long int n);
long double scalbln(long double x, long int n); // see 20.2
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);

float cbrt(float x); // see 20.2
double cbrt(double x);
long double cbrt(long double x); // see 20.2
float cbrtf(float x);
long double cbrtl(long double x);

// 29.9.1, 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 20.2
double fabs(double x);
long double fabs(long double x); // see 20.2
float fabsf(float x);
long double fabsl(long double x);

float hypot(float x, float y); // see 20.2
double hypot(double x, double y);
long double hypot(long double x, long double y); // see 20.2
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

// 29.9.3, 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 20.2
double pow(double x, double y);
long double pow(long double x, long double y); // see 20.2
float powf(float x, float y);
long double powl(long double x, long double y);

float sqrt(float x); // see 20.2
double sqrt(double x);
long double sqrt(long double x); // see 20.2
float sqrtf(float x);
long double sqrtl(long double x);

float erf(float x); // see 20.2
double erf(double x);
long double erf(long double x); // see 20.2
float erff(float x);
long double erfl(long double x);

float erfc(float x); // see 20.2
double erfc(double x);
long double erfc(long double x); // see 20.2
float erfcf(float x);
long double erfcl(long double x);

float lgamma(float x); // see 20.2
double lgamma(double x);
long double lgamma(long double x); // see 20.2
float lgammaf(float x);
long double lgammal(long double x);

float tgamma(float x); // see 20.2
double tgamma(double x);
long double tgamma(long double x); // see 20.2
float tgammaf(float x);
long double tgammal(long double x);

float ceil(float x); // see 20.2
double ceil(double x);
long double ceil(long double x); // see 20.2
float ceilf(float x);
long double ceill(long double x);

float floor(float x); // see 20.2
double floor(double x);
long double floor(long double x); // see 20.2
float floorf(float x);
long double floorl(long double x);
float nearbyint(float x); // see 20.2
double nearbyint(double x);
long double nearbyint(long double x); // see 20.2
float nearbyintf(float x);
long double nearbyintl(long double x);

float rint(float x); // see 20.2
double rint(double x);
long double rint(long double x); // see 20.2
float rintf(float x);
long double rintl(long double x);

long int lrint(float x); // see 20.2
long int lrint(double x);
long int lrint(long double x); // see 20.2
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(float x); // see 20.2
long long int llrint(double x);
long long int llrint(long double x); // see 20.2
long long int llrintf(float x);
long long int llrintl(long double x);

float round(float x); // see 20.2
double round(double x);
long double round(long double x); // see 20.2
float roundf(float x);
long double roundl(long double x);

long int lround(float x); // see 20.2
long int lround(double x);
long int lround(long double x); // see 20.2
long int lroundf(float x);
long int lroundl(long double x);

long long int llround(float x); // see 20.2
long long int llround(double x);
long long int llround(long double x); // see 20.2
long long int llroundf(float x);
long long int llroundl(long double x);

float trunc(float x); // see 20.2
double trunc(double x);
long double trunc(long double x); // see 20.2
float truncf(float x);
long double truncl(long double x);

float fmod(float x, float y); // see 20.2
double fmod(double x, double y);
long double fmod(long double x, long double y); // see 20.2
float fmodf(float x, float y);
long double fmodl(long double x, long double y);

float remainder(float x, float y); // see 20.2
double remainder(double x, double y);
long double remainder(long double x, long double y); // see 20.2
float remainderf(float x, float y);
long double remaindrl(long double x, long double y);

float remquo(float x, float y, int* quo); // see 20.2
double remquo(double x, double y, int* quo);
long double remquo(long double x, long double y, int* quo); // see 20.2
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 20.2
double copysign(double x, double y);
long double copysign(long double x, long double y); // see 20.2
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 20.2
double nextafter(double x, double y);
long double nextafter(long double x, long double y); // see 20.2
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

float nexttoward(float x, long double y); // see 20.2
double nexttoward(double x, long double y);
long double nexttoward(long double x, long double y); // see 20.2
float nexttowardf(float x, float y);
long double nexttowardl(long double x, long double y);

float fdim(float x, float y); // see 20.2
double fdim(double x, double y);
long double fdim(long double x, long double y); // see 20.2
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

float fmax(float x, float y); // see 20.2
double fmax(double x, double y);
long double fmax(long double x, long double y); // see 20.2
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

float fmin(float x, float y); // see 20.2
double fmin(double x, double y);
long double fmin(long double x, long double y); // see 20.2
float fminf(float x, float y);
long double fminl(long double x, long double y);

float fma(float x, float y, float z); // see 20.2
double fma(double x, double y, double z);
long double fma(long double x, long double y, long double z); // see 20.2
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

// 29.9.4, classification / comparison functions
int fpclassify(float x);
int fpclassify(double x);
int fpclassify(long double x);

int isfinite(float x);
int isfinite(double x);
int isfinite(long double x);

int isinf(float x);
int isinf(double x);
int isinf(long double x);

int isnan(float x);
int isnan(double x);
int isnan(long double x);
int isnormal(float x);
int isnormal(double x);
int isnormal(long double x);

int signbit(float x);
int signbit(double x);
int signbit(long double x);

int isgreater(float x, float y);
int isgreater(double x, double y);
int isgreater(long double x, long double y);

int isgreaterequal(float x, float y);
int isgreaterequal(double x, double y);
int isgreaterequal(long double x, long double y);

int isless(float x, float y);
int isless(double x, double y);
int isless(long double x, long double y);

int islessequal(float x, float y);
int islessequal(double x, double y);
int islessequal(long double x, long double y);

int islessgreater(float x, float y);
int islessgreater(double x, double y);
int islessgreater(long double x, long double y);

int isunordered(float x, float y);
int isunordered(double x, double y);
int isunordered(long double x, long double y);

// 29.9.5, mathematical special functions

// 29.9.5.1, 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);

// 29.9.5.2, 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);

// 29.9.5.3, beta function
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// 29.9.5.4, 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);

// 29.9.5.5, 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);

// 29.9.5.6, 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);
// 29.9.5.7, 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);

// 29.9.5.8, 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);

// 29.9.5.9, 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);

// 29.9.5.10, 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);

// 29.9.5.11, 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);

// 29.9.5.12, 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);

// 29.9.5.13, 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);

// 29.9.5.14, exponential integral
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// 29.9.5.15, Hermite polynomials
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

// 29.9.5.16, Laguerre polynomials
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// 29.9.5.17, Legendre polynomials
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// 29.9.5.18, Riemann zeta function
double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double x);

// 29.9.5.19, spherical Bessel functions of the first kind
double sph_bessel(unsigned n, double x);
float sph_besself(unsigned n, float x);
The contents and meaning of the header `<cmath>` are the same as the C standard library header `<math.h>`, with the addition of a three-dimensional hypotenuse function (29.9.3) and the mathematical special functions described in 29.9.5. [Note: Several functions have additional overloads in this document, but they have the same behavior as in the C standard library (20.2). —end note]

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`.
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

### 29.9.2 Absolute values

#### [c.math.abs]

The headers `<cstdlib>` (21.2.2) and `<cmath>` (29.9.1) declare the functions described in this subclause. —end note]

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

**Effects:** The `abs` functions have the semantics specified in the C standard library for the functions `abs`, `labs`, `llabs`, `fabs`, `fabsf`, and `fabsl`.

**Remarks:** If `abs()` is called with an argument of type `X` for which `is_unsigned_v<X>` is `true` and if `X` cannot be converted to `int` by integral promotion (7.6), 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

### 29.9.3 Three-dimensional hypotenuse

#### [c.math.hypot3]

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

**Returns:** $\sqrt{x^2 + y^2 + z^2}$.

### 29.9.4 Classification / comparison functions

#### [c.math.fpclass]

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

§ 29.9.4 1025
29.9.5 Mathematical special functions [sf.cmath]

If any argument value to any of the functions specified in this subclause is a NaN (Not a Number), the
function shall return a NaN but it shall not report a domain error. Otherwise, the function shall report a
domain error for just those argument values for which:

1. the function description’s Returns: clause explicitly specifies a domain and those argument values fall
   outside the specified domain, or
2. the corresponding mathematical function value has a nonzero imaginary component, or
3. the corresponding mathematical function is not mathematically defined.

Unless otherwise specified, each function is defined for all finite values, for negative infinity, and for positive
infinity.

29.9.5.1 Associated Laguerre polynomials [sf.cmath.assoc_laguerre]

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);

Effects: These functions compute the associated Laguerre polynomials of their respective arguments n, m,
and x.

Returns:

\[ L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x), \quad \text{for } x \geq 0 \]

where \( n \) is \( n \), \( m \) is \( m \), and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \) or if \( m \geq 128 \).

29.9.5.2 Associated Legendre functions [sf.cmath.assoc_legendre]

double assoc_legendre(unsigned l, unsigned m, double x);
float assoc_legendref(unsigned l, unsigned m, float x);
long double assoc_legendrel(unsigned l, unsigned m, long double x);

Effects: These functions compute the associated Legendre functions of their respective arguments l, m,
and x.

Returns:

\[ P_l^m(x) = (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_l(x), \quad \text{for } |x| \leq 1 \]

where \( l \) is \( l \), \( m \) is \( m \), and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).

29.9.5.3 Beta function [sf.cmath.beta]

double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

Effects: These functions compute the beta function of their respective arguments \( x \) and \( y \).

Returns:

\[ B(x, y) = \frac{\Gamma(x) \Gamma(y)}{\Gamma(x + y)}, \quad \text{for } x > 0, \ y > 0 \]

where \( x \) is \( x \) and \( y \) is \( y \).

\[ 289 \] 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.
29.9.5.4 Complete elliptic integral of the first kind

\[ \text{comp\_ellint\_1}(k) \]
\[ \text{comp\_ellint\_1f}(k) \]
\[ \text{comp\_ellint\_1l}(k) \]

1. **Effects:** These functions compute the complete elliptic integral of the first kind of their respective arguments \( k \).

2. **Returns:**

\[ K(k) = F(k, \pi/2), \quad \text{for } |k| \leq 1 \]

where \( k \) is \( k \).

See also 29.9.5.11.

29.9.5.5 Complete elliptic integral of the second kind

\[ \text{comp\_ellint\_2}(k) \]
\[ \text{comp\_ellint\_2f}(k) \]
\[ \text{comp\_ellint\_2l}(k) \]

1. **Effects:** These functions compute the complete elliptic integral of the second kind of their respective arguments \( k \).

2. **Returns:**

\[ E(k) = E(k, \pi/2), \quad \text{for } |k| \leq 1 \]

where \( k \) is \( k \).

See also 29.9.5.12.

29.9.5.6 Complete elliptic integral of the third kind

\[ \text{comp\_ellint\_3}(k, \nu) \]
\[ \text{comp\_ellint\_3f}(k, \nu) \]
\[ \text{comp\_ellint\_3l}(k, \nu) \]

1. **Effects:** These functions compute the complete elliptic integral of the third kind of their respective arguments \( k \) and \( \nu \).

2. **Returns:**

\[ \Pi(\nu, k) = \Pi(\nu, k, \pi/2), \quad \text{for } |k| \leq 1 \]

where \( k \) is \( k \) and \( \nu \) is \( \nu \).

See also 29.9.5.13.

29.9.5.7 Regular modified cylindrical Bessel functions

\[ \text{cyl\_bessel\_i}(\nu, x) \]
\[ \text{cyl\_bessel\_if}(\nu, x) \]
\[ \text{cyl\_bessel\_il}(\nu, x) \]

1. **Effects:** These functions compute the regular modified cylindrical Bessel functions of their respective arguments \( \nu \) and \( x \).

2. **Returns:**

\[ I_\nu(x) = i^{-\nu} J_\nu(ix) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k! \Gamma(\nu+k+1)}, \quad \text{for } x \geq 0 \]

where \( \nu \) is \( \nu \) and \( x \) is \( x \).

3. **Remarks:** The effect of calling each of these functions is implementation-defined if \( \nu \geq 128 \).

See also 29.9.5.8.
Cylindrical Bessel functions of the first kind

```c
double cyl_bessel_j(double nu, double x);
float cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);
```

**Effects:** These functions compute the cylindrical Bessel functions of the first kind of their respective arguments $\nu$ and $x$.

**Returns:**

$$ J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k! \Gamma(\nu + k + 1)} , \quad \text{for } x \geq 0 $$

where $\nu$ is $\nu$ and $x$ is $x$.

**Remarks:** The effect of calling each of these functions is implementation-defined if $\nu \geq 128$.

Irregular modified cylindrical Bessel functions

```c
double cyl_bessel_k(double nu, double x);
float cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);
```

**Effects:** These functions compute the irregular modified cylindrical Bessel functions of their respective arguments $\nu$ and $x$.

**Returns:**

$$ K_\nu(x) = \left(\pi/2\right)^{\nu+1} \left( J_\nu(ix) + i N_\nu(ix) \right) = \begin{cases} \frac{\pi}{2} \left( \frac{1}{\nu} \right) \left( \frac{1}{\nu} \right) \cos \nu \pi - \left( \frac{1}{\nu} \right) \left( \frac{1}{\nu} \right) \sin \nu \pi , & \text{for } x \geq 0 \text{ and non-integral } \nu \\
\lim_{\mu \to \nu} \frac{1}{\sin \nu \pi} \left( J_\nu(x) \cos \mu \pi - J_\nu(x) \sin \mu \pi \right) , & \text{for } x \geq 0 \text{ and integral } \nu 
\end{cases} $$

where $\nu$ is $\nu$ and $x$ is $x$.

**Remarks:** The effect of calling each of these functions is implementation-defined if $\nu \geq 128$.

Cylindrical Neumann functions

```c
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 29.9.5.7, 29.9.5.8, 29.9.5.10.

Incomplete elliptic integral of the first kind

```c
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 \( \phi \).

### 29.9.5.12 Incomplete elliptic integral of the second kind

\[ \text{double} \quad \text{ellint}_2(\text{double} \; k, \text{double} \; \phi); \]
\[ \text{float} \quad \text{ellint}_2f(\text{float} \; k, \text{float} \; \phi); \]
\[ \text{long double} \quad \text{ellint}_2l(\text{long double} \; k, \text{long double} \; \phi); \]

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 \( \phi \).

### 29.9.5.13 Incomplete elliptic integral of the third kind

\[ \text{double} \quad \text{ellint}_3(\text{double} \; k, \text{double} \; \nu, \text{double} \; \phi); \]
\[ \text{float} \quad \text{ellint}_3f(\text{float} \; k, \text{float} \; \nu, \text{float} \; \phi); \]
\[ \text{long double} \quad \text{ellint}_3l(\text{long double} \; k, \text{long double} \; \nu, \text{long double} \; \phi); \]

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 \( \phi \).

### 29.9.5.14 Exponential integral

\[ \text{double} \quad \text{expint}(\text{double} \; x); \]
\[ \text{float} \quad \text{expintf}(\text{float} \; x); \]
\[ \text{long double} \quad \text{expintl}(\text{long double} \; x); \]

Effects: These functions compute the exponential integral of their respective arguments \( x \).

Returns:
\[ \text{Ei}(x) = -\int_{-\infty}^x \frac{e^{-t}}{t} \, dt \]

where \( x \) is \( x \).

### 29.9.5.15 Hermite polynomials

\[ \text{double} \quad \text{hermite}(\text{unsigned} \; n, \text{double} \; x); \]
\[ \text{float} \quad \text{hermitef}(\text{unsigned} \; n, \text{float} \; x); \]
\[ \text{long double} \quad \text{hermitel}(\text{unsigned} \; n, \text{long double} \; x); \]

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 \( n \geq 128 \).
29.9.5.16 Laguerre polynomials

\[ L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x}), \quad \text{for } x \geq 0 \]

where \( n \) is \( n \) and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

29.9.5.17 Legendre polynomials

\[ P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l, \quad \text{for } |x| \leq 1 \]

where \( l \) is \( l \) and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( l \geq 128 \).

29.9.5.18 Riemann zeta function

\[ \zeta(x) = \begin{cases} \sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\ \frac{1}{1 - 2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \leq x \leq 1 \\ 2^x \pi^{x-1} \sin \left( \frac{\pi x}{2} \right) \Gamma(1-x) \zeta(1-x), & \text{for } x < 0 \end{cases} \]

where \( x \) is \( x \).

29.9.5.19 Spherical Bessel functions of the first kind

\[ j_n(x) = (\pi/2x)^{1/2} J_{n+1/2}(x), \quad \text{for } x \geq 0 \]

where \( n \) is \( n \) and \( x \) is \( x \).

Remarks: The effect of calling each of these functions is implementation-defined if \( n \geq 128 \).

See also 29.9.5.8.
### 29.9.5.20 Spherical associated Legendre functions

- **double** `sph_legendre(unsigned l, unsigned m, double theta);`
- **float** `sph_legendref(unsigned l, unsigned m, float theta);`
- **long double** `sph_legendrel(unsigned l, unsigned m, long double theta);`

**Effects:** These functions compute the spherical associated Legendre functions of their respective arguments `l`, `m`, and `theta` (theta measured in radians).

**Returns:**

\[ Y^m_\ell(\theta,0) \]

where

\[ Y^m_\ell(\theta,\phi) = (-1)^m \left( \frac{2\ell + 1}{4\pi} \frac{(\ell - m)!}{(\ell + m)!} \right)^{1/2} P^m_\ell(\cos\theta) e^{im\phi}, \text{ for } |m| \leq \ell \]

and `l` is `l`, `m` is `m`, and `theta` is `theta`.

**Remarks:** The effect of calling each of these functions is implementation-defined if `l >= 128`.

See also 29.9.5.2.

### 29.9.5.21 Spherical Neumann functions

- **double** `sph_neumann(unsigned n, double x);`
- **float** `sph_neumannf(unsigned n, float x);`
- **long double** `sph_neumannl(unsigned n, long double x);`

**Effects:** These functions compute the spherical Neumann functions, also known as the spherical Bessel functions of the second kind, of their respective arguments `n` and `x`.

**Returns:**

\[ n_n(x) = (\pi/2x)^{1/2} N_{n+\frac{1}{2}}(x), \text{ for } x \geq 0 \]

where `n` is `n` and `x` is `x`.

**Remarks:** The effect of calling each of these functions is implementation-defined if `n >= 128`.

See also 29.9.5.10.
30 Input/output library

30.1 General

This Clause describes components that C++ programs may use to perform input/output operations.

The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 106.

Table 106 — Input/output library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.2 Requirements</td>
<td>&lt;iosfwd&gt;</td>
</tr>
<tr>
<td>30.3 Forward declarations</td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td>30.4 Standard iostream objects</td>
<td>&lt;ios&gt;</td>
</tr>
<tr>
<td>30.5 Iostreams base classes</td>
<td>&lt;streambuf&gt;</td>
</tr>
<tr>
<td>30.6 Stream buffers</td>
<td>&lt;streambuf&gt;</td>
</tr>
<tr>
<td>30.7 Formatting and manipulators</td>
<td>&lt;iomanip&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;iostream&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;ostream&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;iomanip&gt;</td>
</tr>
<tr>
<td>30.8 String streams</td>
<td>&lt;sstream&gt;</td>
</tr>
<tr>
<td>30.9 File streams</td>
<td>&lt;fstream&gt;</td>
</tr>
<tr>
<td>30.10 Synchronized output streams</td>
<td>&lt;fstream&gt;</td>
</tr>
<tr>
<td>30.11 File systems</td>
<td>&lt;filesystem&gt;</td>
</tr>
<tr>
<td>30.12 C library files</td>
<td>&lt;cstdio&gt;</td>
</tr>
</tbody>
</table>

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.

30.2 Iostreams requirements

No function described in Clause 30 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 30 or as an overriding virtual function of any class declared in Clause 30 calls imbue, the behavior is undefined.
30.2.2 Positioning type limitations

1 The classes of Clause 30 with template arguments charT and traits behave as described if traits::pos_type and traits::off_type are streampos and streamoff respectively. Except as noted explicitly below, their behavior when traits::pos_type and traits::off_type are other types is implementation-defined.

2 In the classes of Clause 30, 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.

30.2.3 Thread safety

1 Concurrent access to a stream object (30.8, 30.9), stream buffer object (30.6), or C Library stream (30.12) by multiple threads may result in a data race (6.8.2) unless otherwise specified (30.4). [Note: Data races result in undefined behavior (6.8.2). —end note]

2 If one thread makes a library call a that writes a value to a stream and, as a result, another thread reads this value from the stream through a library call b such that this does not result in a data race, then a’s write synchronizes with b’s read.

30.3 Forward declarations

30.3.1 Header <iosfwd> synopsis

namespace std {
    template<class charT> class char_traits;
    template<> class char_traits<char>;
    template<> class char_traits<char16_t>;
    template<> class char_traits<char32_t>;
    template<> class char_traits<wchar_t>;
    template<class T> class allocator;
    template<class charT, class traits = char_traits<charT>>
        class basic_ios;
    template<class charT, class traits = char_traits<charT>>
        class basic_streambuf;
    template<class charT, class traits = char_traits<charT>>
        class basic_istream;
    template<class charT, class traits = char_traits<charT>>
        class basic_ostream;
    template<class charT, class traits = char_traits<charT>>
        class basic_iostream;
    template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
        class basic_stringbuf;
    template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
        class basic_istringstream;
    template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
        class basic_ostringstream;
    template<class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT>>
        class basic_stringstream;
    template<class charT, class traits = char_traits<charT>>
        class 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>
class istreambuf_iterator;
template<class charT, class traits = char_traits<charT>,
         class ostreambuf_iterator>
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 istreamstream = basic_istreamstream<char>;
using ostringstreamstream = basic_ostringstreamstream<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 synccbuf = basic_synccbuf<char>;
using osynccstream = basic_osynccstream<char>;

using vstreambuf = basic_streambuf<wchar_t>;
using vistream = basic_vistream<wchar_t>;
using vostream = basic_vostream<wchar_t>;
using viostream = basic_viostream<wchar_t>;

using wstringbuf = basic_stringbuf<wchar_t>;
using wistreamstream = basic_wistreamstream<wchar_t>;
using wostreamstream = basic_wostreamstream<wchar_t>;
using wstringstream = basic_wstringstream<wchar_t>;

using wfilebuf = basic_filebuf<wchar_t>;
using wifstream = basic_wifstream<wchar_t>;
using wofstream = basic_wofstream<wchar_t>;
using wfstream = basic_wfstream<wchar_t>;

using wsynccbuf = basic_wsynccbuf<wchar_t>;
using wosynccstream = basic_wosynccstream<wchar_t>;

template<class state> class fpos;
using streampos = fpos<char_traits<char>::state_type>;
using wstreampos = fpos<char_traits<wchar_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.\textsuperscript{290}

\textsuperscript{290} 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.
30.3.2 Overview

The class template specialization `basic_ios<charT, traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iosstream` is a class template derived from both `basic_istream<charT, traits>` and `basic_ostream<charT, traits>`. The class template specialization `basic_streambuf<charT, traits>` serves as a base class for class templates `basic_stringbuf` and `basic_filebuf`. The class template specialization `basic_istream<charT, traits>` serves as a base class for class templates `basic_istringstream` and `basic_ifstream`. The class template specialization `basic_ostream<charT, traits>` serves as a base class for class templates `basic_ostringstream` and `basic_ofstream`. The class template specialization `basic_iostream<charT, traits>` serves as a base class for class templates `basic_stringstream` and `basic_fstream`. Other typedef-names define instances of class templates specialized for `char` or `wchar_t` types. Specializations of the class template `fpos` are used for specifying file position information. The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively.

[Note: This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be]

```cpp
template<class stateT> class fpos { ... }; // depends on nothing
using _STATE = ... ; // implementation private declaration of stateT

using streampos = fpos<_STATE>;

template<> struct char_traits<char> {
    using pos_type = streampos;
}

—end note]

30.4 Standard iostream objects

30.4.1 Header `<iostream>` synopsis

```cpp
#include <ios> // see 30.5.1
#include <streambuf> // see 30.6.1
#include <istream> // see 30.7.1
#include <ostream> // see 30.7.2

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;
}
```

30.4.2 Overview

In this Clause, the type name `FILE` refers to the type `FILE` declared in `<cstdio>` (30.12.1). The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (30.12), and includes all the headers necessary to use these objects. The objects are constructed and the associations are established at some time prior to or during the first time an object of class `ios_base::Init` is constructed, and in any case before the body of `main` (6.8.3.1)
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.

Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on FILEs, as specified in the C standard library.

Concurrent access to a synchronized (30.5.3.4) standard iostream object’s formatted and unformatted input (30.7.4.1) and output (30.7.5.1) 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

30.4.3 Narrow stream objects

```cpp
istream cin;
1 The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (30.12.1).
2 After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (30.5.5.2).
```

```cpp
ostream cout;
3 The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (30.12.1).
```

```cpp
ostream cerr;
4 The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (30.12.1).
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` (30.5.5.2).
```

```cpp
ostream clog;
6 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (30.12.1).
```

30.4.4 Wide stream objects

```cpp
wistream wcin;
1 The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>` (30.12.1).
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` (30.5.5.2).
```

```cpp
wostream wcout;
3 The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (30.12.1).
```

```cpp
wostream wcerr;
4 The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (30.12.1).
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` (30.5.5.2).
```

```cpp
wostream wclog;
6 The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (30.12.1).
```

291) If it is possible for them to do so, implementations should initialize the objects earlier than required.
292) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

§ 30.4.4
30.5 Iostreams base classes

30.5.1 Header <ios> synopsis

#include <iosfwd>  // see 30.3.1

namespace std {
    using streamoff = implementation-defined;
    using streamsize = implementation-defined;
    template<class stateT> class fpos;

    class ios_base;
    template<class charT, class traits = char_traits<charT>>
    class basic_ios;

    // 30.5.6, 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);

    // 30.5.6.2, adjustfield
    ios_base& internal (ios_base& str);
    ios_base& left (ios_base& str);
    ios_base& right (ios_base& str);

    // 30.5.6.3, basefield
    ios_base& dec (ios_base& str);
    ios_base& hex (ios_base& str);
    ios_base& oct (ios_base& str);

    // 30.5.6.4, 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);

    // 30.5.6.5, 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;
}
30.5.2 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.\(^{293}\)

using streamsize = implementation-defined;

2 The type streamsize is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.\(^{294}\)

30.5.3 Class ios_base

namespace std {
    class ios_base {
        public:
            class failure; // see below

            // 30.5.3.1.2, 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;

            // 30.5.3.1.3, 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;

            // 30.5.3.1.4, 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;

            // 30.5.3.1.5, seekdir
            using seekdir = T4;
            static constexpr seekdir beg = unspecified;
            static constexpr seekdir cur = unspecified;
            static constexpr seekdir end = unspecified;

            293) Typically long long.
            294) streamsize is used in most places where ISO C would use size_t. Most of the uses of streamsize could use size_t, except for the stringstream constructors, which require negative values. It should probably be the signed type corresponding to size_t (which is what Posix.2 calls ssize_t).
class Init;

    // 30.5.3.2, fmtflags state
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);

    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
    streamsize width(streamsize wide);

    // 30.5.3.3, locales
    locale imbue(const locale& loc);
    locale getloc() const;

    // 30.5.3.5, storage
    static int xalloc();
    long& iword(int index);
    void*& pword(int index);

    // destructor
    virtual ~ios_base();

    // 30.5.3.6, callbacks
    enum event { erase_event, imbue_event, copyfmt_event };
    using event_callback = void (*)(event, ios_base&, int index);
    void register_callback(event_callback fn, int index);
(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

30.5.3.1 Types

30.5.3.1.1 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.

30.5.3.1.2 Type ios_base::fmtflags

using fmtflags = T1;

1 The type fmtflags is a bitmask type (20.4.2.1.4). Setting its elements has the effects indicated in Table 107.

2 Type fmtflags also defines the constants indicated in Table 108.

30.5.3.1.3 Type ios_base::iostate

using iostate = T2;

1 The type iostate is a bitmask type (20.4.2.1.4) that contains the elements indicated in Table 109.

2 Type iostate also defines the constant:

(2.1) — goodbit, the value zero.

30.5.3.1.4 Type ios_base::openmode

using openmode = T3;

1 The type openmode is a bitmask type (20.4.2.1.4). It contains the elements indicated in Table 110.

30.5.3.1.5 Type ios_base::seekdir

using seekdir = T4;

1 The type seekdir is an enumerated type (20.4.2.1.3) that contains the elements indicated in Table 111.
### Table 107 — fmtflags effects

<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>skips</td>
<td>skips leading whitespace before certain input operations</td>
</tr>
<tr>
<td>unitbuf</td>
<td>flushes output after each output operation</td>
</tr>
<tr>
<td>uppercase</td>
<td>replaces certain lowercase letters with their uppercase equivalents in generated output</td>
</tr>
</tbody>
</table>

### Table 108 — 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>

### Table 109 — 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>

### Table 110 — 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>

### Table 111 — 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>
30.5.3.1.6 Class ios_base::Init

namespace std {
    class ios_base::Init {
    public:
        Init();
        ~Init();
    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>` (30.4) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (30.12.1).

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().

30.5.3.2 ios_base state functions

fmtflags flags() const;

Returns: The format control information for both input and output.

fmtflags flags(fmtflags fmtfl);

Postconditions: 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);

Postconditions: 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);

Postconditions: wide == width().
Returns: The previous value of \texttt{width()}.

30.5.3.3 \texttt{ios\_base} functions

\begin{verbatim}
locale imbue(const locale& loc);
\end{verbatim}

\textbf{Effects:} Calls each registered callback pair \((fn, index)\) (30.5.3.6) as \((*fn)(imbue\_event, \ast this, index)\) at such a time that a call to \texttt{ios\_base::getloc()} from within \texttt{fn} returns the new locale value \texttt{loc}.

\textbf{Returns:} The previous value of \texttt{getloc()}. 

\textbf{Postconditions:} \texttt{loc == getloc()}. 

\begin{verbatim}
locale getloc() const;
\end{verbatim}

\textbf{Returns:} If no locale has been imbued, a copy of the global C++ locale, \texttt{locale()}, in effect at the time of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input and output operations.

30.5.3.4 \texttt{ios\_base} static members

\begin{verbatim}
bool sync_with_stdio(bool sync = true);
\end{verbatim}

\textbf{Returns:} \texttt{true} if the previous state of the standard iostream objects (30.4) was synchronized and otherwise returns \texttt{false}. The first time it is called, the function returns \texttt{true}.

\textbf{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 \texttt{false} argument, it allows the standard streams to operate independently of the standard C streams.

When a standard iostream object \texttt{str} is \textit{synchronized} with a standard stdio stream \texttt{f}, the effect of inserting a character \texttt{c} by

\begin{verbatim}
fputc(f, c);
\end{verbatim}

is the same as the effect of

\begin{verbatim}
str.rdbuf()->sputc(c);
\end{verbatim}

for any sequences of characters; the effect of extracting a character \texttt{c} by

\begin{verbatim}
c = fgetc(f);
\end{verbatim}

is the same as the effect of

\begin{verbatim}
c = str.rdbuf()->sbumpc();
\end{verbatim}

for any sequences of characters; and the effect of pushing back a character \texttt{c} by

\begin{verbatim}
ungetc(c, f);
\end{verbatim}

is the same as the effect of

\begin{verbatim}
str.rdbuf()->sputbackc(c);
\end{verbatim}

for any sequence of characters.\footnote{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.}

30.5.3.5 \texttt{ios\_base} storage functions

\begin{verbatim}
static int xalloc();
\end{verbatim}

\textbf{Returns:} \texttt{index ++}. 

\textbf{Remarks:} Concurrent access to this function by multiple threads shall not result in a data race (6.8.2).

\begin{verbatim}
long& iword(int idx);
\end{verbatim}

\textbf{Effects:} If \texttt{iarray} is a null pointer, allocates an array of \texttt{long} of unspecified size and stores a pointer to its first element in \texttt{iarray}. The function then extends the array pointed at by \texttt{iarray} as necessary 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 `copyfmt()`, calling `ivword` with the same index yields another reference to the same value. If the function fails and `*this` is a base class subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

Returns: On success `iarray[idx]`. On failure, a valid `long&` initialized to 0.

```cpp
void*& pword(int idx);
```

Effects: If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `iarray[idx]`. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to `copyfmt()`, calling `pword` with the same index yields another reference to the same value. If the function fails and `*this` is a base class subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

Returns: On success `parray[idx]`. On failure a valid `void*&` initialized to 0.

Remarks: After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

### 30.5.3.6 `ios_base` callbacks

```cpp
void register_callback(event_callback fn, int index);
```

Effects: Registers the pair `(fn, index)` such that during calls to `imbue()` (30.5.3.3), `copyfmt()`, or `~ios_base()` (30.5.3.7), the function `fn` is called with argument `index`. 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.

Requires: The function `fn` shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

### 30.5.3.7 `ios_base` constructors/destructor

```cpp
ios_base();
```

Effects: Each `ios_base` member has an indeterminate value after construction. The object’s members shall be initialized by calling `basic_ios::init` before the object’s first use or before it is destroyed, whichever comes first; otherwise the behavior is undefined.

```cpp
~ios_base();
```

Effects: Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` (30.5.3.6) as `(*fn)(erase_event, *this, index)` at such time that any `ios_base` member function called from within `fn` has well-defined results.

### 30.5.4 Class template `fpos`

```cpp
namespace std {
    template<class stateT> class fpos {
        public:
            // 30.5.4.1, members
            stateT state() const;
            void state(stateT);
        private;
            stateT st; // exposition only
    };
}
```

296) An implementation is free to implement both the integer array pointed at by `iarray` and the pointer array pointed at by `parray` as sparse data structures, possibly with a one-element cache for each.

297) For example, because it cannot allocate space.

298) For example, because it cannot allocate space.
30.5.4.1 fpos members

void state(stateT s);

Effects: Assigns s to st.

stateT state() const;

Returns: Current value of st.

30.5.4.2 fpos requirements

Operations specified in Table 112 are permitted. In that table,

1. P refers to an instance of fpos,
2. p and q refer to values of type P,
3. 0 refers to type streamoff,
4. o refers to a value of type streamoff,
5. sz refers to a value of type streamsize and
6. i refers to a value of type int.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Operational semantics</th>
<th>Assertion/note</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(i)</td>
<td>p</td>
<td>p == P(i)</td>
<td>note: a destructor is assumed.</td>
</tr>
<tr>
<td>P p(i);</td>
<td>P p = i;</td>
<td>Postconditions: p == P(i).</td>
<td></td>
</tr>
<tr>
<td>P(o)</td>
<td>fpos</td>
<td>converts from offset</td>
<td></td>
</tr>
<tr>
<td>0(p)</td>
<td>streamoff</td>
<td>converts to offset</td>
<td></td>
</tr>
<tr>
<td>p == q</td>
<td>convertible to bool</td>
<td>== is an equivalence relation</td>
<td></td>
</tr>
<tr>
<td>p != q</td>
<td>convertible to bool</td>
<td>!(p == q)</td>
<td></td>
</tr>
<tr>
<td>q = p + o</td>
<td>fpos</td>
<td>+ offset</td>
<td></td>
</tr>
<tr>
<td>p ++ o</td>
<td>fpos</td>
<td>- offset</td>
<td></td>
</tr>
<tr>
<td>q = p - o</td>
<td>fpos</td>
<td>distance</td>
<td></td>
</tr>
<tr>
<td>p -= o</td>
<td>fpos</td>
<td></td>
<td></td>
</tr>
<tr>
<td>o = p - q</td>
<td>streamoff</td>
<td></td>
<td></td>
</tr>
<tr>
<td>streamsize(o)</td>
<td>streamsize</td>
<td>converts streamsize(O(sz)) == sz</td>
<td></td>
</tr>
<tr>
<td>O(sz)</td>
<td>streamoff</td>
<td>converts streamsize(O(sz)) == sz</td>
<td></td>
</tr>
</tbody>
</table>

2 [Note: Every implementation is required to supply overloaded operators on fpos objects to satisfy the requirements of 30.5.4.2. It is unspecified whether these operators are members of fpos, global operators, or provided in some other way. —end note]

3 Stream operations that return a value of type traits::pos_type return P(O(-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.

30.5.5 Class template basic_ios

30.5.5.1 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;

§ 30.5.5.1
// 30.5.5.4, flags functions
explicit operator bool() const;
bool operator!() const;
iosstate rdstate() const;
void clear(iosstate state = goodbit);
void setstate(iosstate state);
bool good() const;
bool eof() const;
bool fail() const;
bool bad() const;
iosstate exceptions() const;
void exceptions(iosstate except);

// 30.5.5.2, constructor/destructor
explicit basic_ios(basic_streambuf<charT, traits>* sb);
virtual ~basic_ios();

// 30.5.5.3, 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);
};

30.5.5.2 basic_ios constructors

explicit basic_ios(basic_streambuf<charT, traits>* sb);

1 Effects: Constructs an object of class basic_ios, assigning initial values to its member objects by calling init(sb).

basic_ios();

2 Effects: Constructs an object of class basic_ios (30.5.3.7) 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.

~basic_ios();

3 Remarks: The destructor does not destroy rdbuf().
void init(basic_streambuf<charT, traits>* sb);

Postconditions: The postconditions of this function are indicated in Table 113.

Table 113 — basic_ios::init() effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbuf()</td>
<td>sb</td>
</tr>
<tr>
<td>tie()</td>
<td>0</td>
</tr>
<tr>
<td>rdstate()</td>
<td>goodbit if sb is not a null pointer, otherwise badbit.</td>
</tr>
<tr>
<td>exceptions()</td>
<td>goodbit</td>
</tr>
<tr>
<td>flags()</td>
<td>skipws</td>
</tr>
<tr>
<td>width()</td>
<td>0</td>
</tr>
<tr>
<td>precision()</td>
<td>6</td>
</tr>
<tr>
<td>fill()</td>
<td>widen(' ')</td>
</tr>
<tr>
<td>getloc()</td>
<td>a copy of the value returned by locale()</td>
</tr>
<tr>
<td>iarray</td>
<td>a null pointer</td>
</tr>
<tr>
<td>parray</td>
<td>a null pointer</td>
</tr>
</tbody>
</table>

30.5.5.3 Member functions

basic_ostream<charT, traits>* tie() const;

Returns: An output sequence that is tied to (synchronized with) the sequence controlled by the stream buffer.

basic_ostream<charT, traits>* tie(basic_ostream<charT, traits>* tiestr);

Requires: If tiestr is not null, tiestr shall not be reachable by traversing the linked list of tied stream objects starting from tiestr->tie().

Postconditions: tiestr == tie().

Returns: The previous value of tie().

basic_streambuf<charT, traits>* rdbuf() const;

Returns: A pointer to the streambuf associated with the stream.

basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT, traits>* sb);

Postconditions: sb == rdbuf().

Effects: Calls clear().

Returns: The previous value of rdbuf().

locale imbue(const locale& loc);

Effects: Calls ios_base::imbue(loc) (30.5.3.3) and if rdbuf() != 0 then rdbuf()->pubimbue(loc) (30.6.3.2.1).

Returns: The prior value of ios_base::imbue().

cchar narrow(char_type c, char default) const;

Returns: use_facet<ctype<char_type>>(getloc()).narrow(c, default)

cchar widen(char c) const;

Returns: use_facet<ctype<char_type>>(getloc()).widen(c)

cchar fill() const;

Returns: The character used to pad (fill) an output conversion to the specified field width.

cchar fill(char_type fillch);

Postconditions: traits::eq(fillch, fill()).
Returns: The previous value of `fill()`.

```cpp
basic_ios& copyfmt(const basic_ios& rhs);
```

Effects: If `(this == &rhs)` does nothing. Otherwise assigns to the member objects of `*this` the corresponding member objects of `rhs` as follows:

- calls each registered callback pair `(fn, index)` as `(*fn)(erase_event, *this, index);
- then, assigns to the member objects of `*this` the corresponding member objects of `rhs`, except that
-  - `rdstate()`, `rdbuf()`, and `exceptions()` are left unchanged;
-  - the contents of arrays pointed at by `pword` and `iword` are copied, not the pointers themselves; and
-  - 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;
- then, calls each callback pair that was copied from `rhs` as `(*fn)(copyfmt_event, *this, index);
- 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]

Postconditions: The postconditions of this function are indicated in Table 114.

Table 114 — `basic_ios::copyfmt()` effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdbuf()</td>
<td>unchanged</td>
</tr>
<tr>
<td>tie()</td>
<td>rhs.tie()</td>
</tr>
<tr>
<td>rdstate()</td>
<td>unchanged</td>
</tr>
<tr>
<td>exceptions()</td>
<td>rhs.exceptions()</td>
</tr>
<tr>
<td>flags()</td>
<td>rhs.flags()</td>
</tr>
<tr>
<td>width()</td>
<td>rhs.width()</td>
</tr>
<tr>
<td>precision()</td>
<td>rhs.precision()</td>
</tr>
<tr>
<td>fill()</td>
<td>rhs.fill()</td>
</tr>
<tr>
<td>getloc()</td>
<td>rhs.getloc()</td>
</tr>
</tbody>
</table>

Returns: `*this`.

```cpp
void move(basic_ios& rhs);
void move(basic_ios&& rhs);
```

Postconditions: `*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.

```cpp
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.

```cpp
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()`.

Postconditions: `rdbuf() == sb`.

Throws: Nothing.

299) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is nonzero.
### 30.5.5.4 basic_ios flags functions

#### explicit operator bool() const;

Returns: !fail().

#### bool operator!() const;

Returns: fail().

#### iostate rdstate() const;

Returns: The error state of the stream buffer.

#### void clear(iostate state = goodbit);

Postconditions: If rdbuf() != 0 then state == rdstate(); otherwise rdstate() == (state | ios_base::badbit).

Effects: If ((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0, returns. Otherwise, the function throws an object of class basic_ios::failure (30.5.3.1.1), constructed with implementation-defined argument values.

#### void setstate(iostate state);

Effects: Calls clear(rdstate() | state) (which may throw basic_ios::failure (30.5.3.1.1)).

#### 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().\(^{300}\)

#### bool bad() const;

Returns: true if badbit is set in rdstate().

#### iostate exceptions() const;

Returns: A mask that determines what elements set in rdstate() cause exceptions to be thrown.

#### void exceptions(iostate except);

Postconditions: except == exceptions().

Effects: Calls clear(rdstate()).

### 30.5.6 ios_base manipulators

#### 30.5.6.1 fmtflags manipulators

#### 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.

\(^{300}\) Checking badbit also for fail() is historical practice.
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.

30.5.6.2 adjustfield manipulators [adjustfield.manip]

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.

30.5.6.3 basefield manipulators

ios_base& dec(ios_base& str);
   Effects: Calls str.setf(ios_base::dec, ios_base::basefield).
   Returns: str\(^{301}\).

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.

30.5.6.4 floatfield manipulators

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.

[Note: The more obvious use of `ios_base::hex` to specify hexadecimal floating-point format would change the meaning of existing well-defined programs. C++ 2003 gives no meaning to the combination of `fixed` and `scientific`. —end note]

ios_base& defaultfloat(ios_base& str);
   Effects: Calls str.unsetf(ios_base::floatfield).
   Returns: str.

30.5.6.5 Error reporting

error_code make_error_code(io_errc e) noexcept;
   Returns: error_code(static_cast<int>(e), iostream_category()).

error_condition make_error_condition(io_errc e) noexcept;
   Returns: error_condition(static_cast<int>(e), iostream_category()).

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".

\(^{301}\) The function signature `dec(ios_base&)` can be called by the function signature `basic_ostream<ostream>::operator<<(ios_base& (*)(ios_base&))` to permit expressions of the form `cout << dec` to change the format flags stored in `cout`. 
30.6 Stream buffers

30.6.1 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.

30.6.2 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:

1. If xnext is not a null pointer, then xbeg and xend shall also be non-null pointers into the same charT array, as described above; otherwise, xbeg and xend shall also be null.
2. If xnext is not a null pointer and xnext < xend for an output sequence, then a write position is available. In this case, *xnext shall be assignable as the next element to write (to put, or to store a character value, into the sequence).
3. If xnext is not a null pointer and xbeg < xnext for an input sequence, then a putback position is available. In this case, xnext[-1] shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.
4. If xnext is not a null pointer and xnext < xend for an input sequence, then a read position is available. In this case, *xnext shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

30.6.3 Class template basic_streambuf

```cpp
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_streambuf {
  public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;
```
virtual ~basic_streambuf();

// 30.6.3.2.1, locales
locale pubimbue(const locale& loc);
locale getloc() const;

// 30.6.3.2.2, 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
// 30.6.3.2.3, get area
streamsize in_avail();
int_type snextc();
int_type sbumpc();
int_type sgetc();
streamsize sgetn(char_type* s, streamsize n);

// 30.6.3.2.4, putback
int_type sputbackc(char_type c);
int_type sungetc();

// 30.6.3.2.5, 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);

// 30.6.3.3.2, get area access
char_type* eback() const;
char_type* gptr() const;
char_type* egptr() const;
void gbump(int n);
void setg(char_type* gbeg, char_type* gnext, char_type* gend);

// 30.6.3.3.3, put area access
char_type* pbase() const;
char_type* pptr() const;
char_type* epptr() const;
void pbump(int n);
void setp(char_type* pbeg, char_type* pend);

// 30.6.3.4.1, locales
virtual void imbue(const locale& loc);

// 30.6.3.4.2, buffer management and positioning
virtual basic_streambuf* setbuf(char_type* s, streamsize n);
virtual pos_type seekoff(off_type off, ios_base::seekdir way,
  ios_base::openmode which
  = ios_base::in | ios_base::out);
virtual pos_type seekpos(pos_type sp,
   ios_base::openmode which
   = ios_base::in | ios_base::out);
virtual int sync();

// 30.6.3.4.3, get area
virtual streamsize shoumanyc();
virtual streamsize xsgetn(char_type* s, streamsize n);
virtual int_type underflow();
virtual int_type uflow();

// 30.6.3.4.4, putback
virtual int_type pbackfail(int_type c = traits::eof());

// 30.6.3.4.5, put area
virtual streamsize xsputn(const char_type* s, streamsize n);
virtual int_type overflow(int_type c = traits::eof());
};

The class template basic_streambuf serves as an abstract base class for deriving various stream buffers whose objects each control two character sequences:

— a character input sequence;
— a character output sequence.

30.6.3.1 basic_streambuf constructors

basic_streambuf();

Effects: Constructs an object of class basic_streambuf<charT, traits> and initializes:

(1.1) — all its pointer member objects to null pointers,
(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.

basic_streambuf(const basic_streambuf& rhs);

Effects: Constructs a copy of rhs.

Postconditions:

(4.1) — eback() == rhs.eback()
(4.2) — gptr() == rhs.gptr()
(4.3) — egptr() == rhs.egptr()
(4.4) — pbase() == rhs.pbase()
(4.5) — pptr() == rhs.pptr()
(4.6) — epptr() == rhs.epptr()
(4.7) — getloc() == rhs.getloc()

~basic_streambuf();

Effects: None.

30.6.3.2 basic_streambuf public member functions

30.6.3.2.1 Locales

locale pubimbue(const locale& loc);

Postconditions: loc == getloc().

Effects: Calls imbue(loc).

[streambuf.cons]

[streambuf.members]

[streambuf.locales]
locale getloc() const;

Returns: Previous value of getloc().

locale getloc() const;

Returns: If pubimbue() has ever been called, then the last value of loc supplied, otherwise the current global locale, locale(), in effect at the time of construction. If called after pubimbue() has been called but before pubimbue has returned (i.e., from within the call of imbue()) then it returns the previous value.

30.6.3.2.2 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().

30.6.3.2.3 Get area

streamsize in_avail();

Returns: If a read position is available, returns egptr() - gptr(). Otherwise returns showmanyc() (30.6.3.4.3).

int_type snextc();

Effects: Calls sbumpc().

Returns: If that function returns traits::eof(), returns traits::eof(). Otherwise, returns sgetc().

int_type sbumpc();

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();

Returns: If the input sequence read position is not available, returns underflow(). Otherwise, returns traits::to_int_type(*gptr()).

streamsize sgetn(char_type* s, streamsize n);

Returns: xsgetn(s, n).

30.6.3.2.4 Putback

int_type sputbackc(char_type c);

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();

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()).
30.6.3.2.5 Put area

int_type sputc(char_type c);

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);

Returns: xsputn(s, n).

30.6.3.3 basic_streambuf protected member functions

30.6.3.3.1 Assignment

basic_streambuf& operator=(const basic_streambuf& rhs);

Effects: Assigns the data members of rhs to *this.

Postconditions:
(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()

Returns: *this.

void swap(basic_streambuf& rhs);

Effects: Swaps the data members of rhs and *this.

30.6.3.3.2 Get area access

char_type* eback() const;

Returns: The beginning pointer for the input sequence.

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);

Postconditions: gbeg == eback(), gnext == gptr(), and gend == egptr().

30.6.3.3.3 Put area access

char_type* pbase() const;

Returns: The beginning pointer for the output sequence.

char_type* pptr() const;

Returns: The next pointer for the output sequence.

char_type* epptr() const;

Returns: The end pointer for the output sequence.
void pbump(int n);

Effects: Adds n to the next pointer for the output sequence.

void setp(char_type* pbeg, char_type* pend);

Postconditions: pbeg == pbase(), pbeg == pptr(), and pend == epptr().

30.6.3.4 basic_streambuf virtual functions

30.6.3.4.1 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.

30.6.3.4.2 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 (30.8.2.4, 30.9.2.4).

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 (30.8.2.4, 30.9.2.4).

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 (30.8.2, 30.9.2).

Default behavior: Returns pos_type(off_type(-1)).

int sync();

Effects: Synchronizes the controlled sequences with the arrays. That is, if pbase() is non-null the characters between pbase() and pptr() are written to the controlled sequence. The pointers may then be reset as appropriate.

Returns: -1 on failure. What constitutes failure is determined by each derived class (30.9.2.4).

Default behavior: Returns zero.

30.6.3.4.3 Get area

streamsize showmanyc();\[303\]

Returns: An estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to underflow() will not return traits::eof() until at least that number of characters have been extracted from the stream. If showmanyc() returns -1, then calls to underflow() or uflow() will fail.\[304\]

Default behavior: Returns zero.

Remarks: Uses traits::eof().

\[303\] The morphemes of showmanyc are “es-how-many-see”, not “show-manic”.  
\[304\] underflow or uflow might fail by throwing an exception prematurely. The intention is not only that the calls will not return eof() but that they will return “immediately”.
streamsize xsgetn(char_type* s, streamsize n);

Effects: Assigns up to n characters to successive elements of the array whose first element is designated by s. The characters assigned are read from the input sequence as if by repeated calls to sbumpc(). Assigning stops when either n characters have been assigned or a call to sbumpc() would return traits::eof().

Returns: The number of characters assigned.\(^3\)\(^0\)\(^5\)

Remarks: Uses traits::eof().

int_type underflow();

Remarks: The public members of basic_streambuf call this virtual function only if gptr() is null or gptr() \(\geq\) egptr()

Returns: traits::to_int_type(c), where c is the first character of the pending sequence, without moving the input sequence position past it. If the pending sequence is null then the function returns traits::eof() to indicate failure.

The pending sequence of characters is defined as the concatenation of

- the empty sequence if gptr() is null, otherwise the characters in (gptr(), egptr()), followed by

- some (possibly empty) sequence of characters read from the input sequence.

The result character is the first character of the pending sequence if it is non-empty, otherwise the next character that would be read from the input sequence.

The backup sequence is the empty sequence if eback() is null, otherwise the characters in (eback(), gptr()).

Effects: The function sets up the gptr() and egptr() such that if the pending sequence is non-empty, then egptr() is non-null and the characters in (gptr(), egptr()) are the characters in the pending sequence, otherwise either gptr() is null or gptr() = egptr().

If eback() and gptr() are non-null then the function is not constrained as to their contents, but the usual backup condition” is that either

- the backup sequence contains at least gptr() - eback() characters, in which case the characters in (eback(), gptr()) agree with the last gptr() - eback() characters of the backup sequence, or

- the characters in (gptr() - n, gptr()) agree with the backup sequence (where n is the length of the backup sequence).

Default behavior: Returns traits::eof().

int_type uflow();

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(*gptr()) and increment the value of the next pointer for the input sequence.

Returns: traits::eof() to indicate failure.

30.6.3.4.4 Putback

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

\(^3\)\(^0\)\(^5\) Classes derived from basic_streambuf can provide more efficient ways to implement xsgetn() and xsputn() by overriding these definitions from the base class.
— If traits::eq_int_type(c, traits::eof()) returns true, then the input sequence is backed up one character before the pending sequence is determined.

— If traits::eq_int_type(c, traits::eof()) returns false, then c is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

Postconditions: On return, the constraints of gptr(), eback(), and pptr() are the same as for underflow().

Returns: traits::eof() to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. pbackfail() is called only when put back has really failed.

Returns some value other than traits::eof() to indicate success.

Default behavior: Returns traits::eof().

30.6.3.4.5 Put area

streamsize xsputn(const char_type* s, streamsize n);

1 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 ppotr() == epptr() becomes true or whether it achieves the same effects by other means.

2 Returns: The number of characters written.

int_type overflow(int_type c = traits::eof());

3 Effects: Consumes some initial subsequence of the characters of the pending sequence. The pending sequence is defined as the concatenation of

— the empty sequence if pbase() is null, otherwise the ppotr() - pbase() characters beginning at pbase(), followed by

— the empty sequence if traits::eq_int_type(c, traits::eof()) returns true, otherwise the sequence consisting of c.

4 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.

5 Requires: Every overriding definition of this virtual function shall obey the following constraints:

— The effect of consuming a character on the associated output sequence is specified.306

— Let r be the number of characters in the pending sequence not consumed. If r is nonzero then pbase() and ppotr() shall be set so that: ppotr() - pbase() == r and the r characters starting at pbase() are the associated output stream. In case r is zero (all characters of the pending sequence have been consumed) then either pbase() is set to nullptr, or pbase() and ppotr() are both set to the same non-null value.

— The function may fail if either appending some character to the associated output stream fails or if it is unable to establish pbase() and ppotr() according to the above rules.

6 Returns: traits::eof() or throws an exception if the function fails. Otherwise, returns some value other than traits::eof() to indicate success.307

Default behavior: Returns traits::eof().

30.7 Formatting and manipulators

30.7.1 Header <istream> synopsis

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_istream;

306) That is, for each class derived from an instance of basic_streambuf in this Clause (30.8.2, 30.9.2), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.

307) Typically, overflow returns c to indicate success, except when traits::eq_int_type(c, traits::eof()) returns true, in which case it returns traits::not_eof(c).
using istream = basic_istream<char>;
using wistream = basic_istream<wchar_t>;

template<class charT, class traits = char_traits<charT>>
    class basic_iostream;

using iostream = basic_iostream<char>;
using wiostream = basic_iostream<wchar_t>;

template<class charT, class traits>
    basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);

template<class charT, class traits, class T>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& is, T&& x);

30.7.2 Header <ostream> synopsis

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, class T>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&& os, const T& x);
}
template<class charT, class traits>
T14 quoted(basic_string_view<charT, traits> s,
    charT delim = charT('"'), charT escape = charT('\'));
}

§ 30.7.4 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.

30.7.4.1 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 (30.5.5))
        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;

        // 30.7.4.1.1, constructor/destructor
        explicit basic_istream(basic_streambuf<charT, traits>* sb);
        virtual ~basic_istream();

        // 30.7.4.1.3, prefix/suffix
        class sentry;

        // 30.7.4.2, formatted input
        basic_istream<charT, traits>&
            operator>>(basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&&));
        basic_istream<charT, traits>&
            operator>>(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&&));
        basic_istream<charT, traits>&
            operator>>(ios_base& (*pf)(ios_base&));
        basic_istream<charT, traits>& operator>>(bool& n);
        basic_istream<charT, traits>& operator>>(short& n);
        basic_istream<charT, traits>& operator>>(unsigned short& n);
        basic_istream<charT, traits>& operator>>(int& n);
        basic_istream<charT, traits>& operator>>(unsigned int& n);
        basic_istream<charT, traits>& operator>>(long& n);
        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);

        // 30.7.4.3, 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>& get(basic_streambuf<char_type, traits>&& sb);
        basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>&& sb, 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);

§ 30.7.4.1 1061
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();
pos_type tellg();
basic_istream<charT, traits>& seekg(pos_type);
basic_istream<charT, traits>& seekg(off_type, ios_base::seekdir);

protected:
    // 30.7.4.1.1, copy/move constructor
    basic_istream(const basic_istream& rhs) = delete;
basic_istream(basic_istream&& rhs);
    // 30.7.4.1.2, assign and swap
    basic_istream& operator=(const basic_istream& rhs) = delete;
basic_istream& operator=(basic_istream&& rhs);
    void swap(basic_istream& rhs);
};

// 30.7.4.2.3, 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&);
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*);

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.

2 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.

3 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 (30.5.5.4), before returning.

4 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.

30.7.4.1.1 basic_istream constructors
explicit basic_istream(basic_streambuf<charT, traits>* sb);

1 Effects: Constructs an object of class basic_istream, initializing the base class subobject with
basic_ios::init(sb) (30.5.5.2).

2 Postconditions: gcount() == 0.
basic_istream(basic_istream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by default constructing the base class, copying the gcount() from rhs, calling basic_ios<charT, traits>::move(rhs) to initialize the base class, and setting the gcount() for rhs to 0.

virtual ~basic_istream();

Effects: Destroys an object of class basic_istream.

Remarks: Does not perform any operations of rdbuf().

30.7.4.1.2 Class basic_istream assign and swap

basic_istream& operator=(basic_istream&& rhs);

Effects: As if by swap(rhs).

Returns: *this.

void swap(basic_istream& rhs);

Effects: Calls basic_ios<charT, traits>::swap(rhs). Exchanges the values returned by gcount() and rhs.gcount().

30.7.4.1.3 Class basic_istream::sentry

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;
};

} // namespace std

The class sentry defines a class that is responsible for doing exception safe prefix and suffix operations.

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.\footnote{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.} If noskipws is zero and is.flags() & ios_base::skipws is nonzero, the function extracts and discards each character as long as the next available input character c is a whitespace character. If is.rdbuf()->sbumpc() or is.rdbuf()->sgetc() returns traits::eof(), the function calls setstate(failbit | eofbit) (which may throw ios_base::failure).

Remarks: The constructor

\begin{verbatim}
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.
\end{verbatim}

To decide if the character c is a whitespace character, the constructor performs as if it executes the following code fragment:

\begin{verbatim}
    const ctype<charT>& ctype = use_facet<ctype<charT>>(is.getloc());
    if (ctype.is(ctype.space, c) != 0)
        // c is a whitespace character.
\end{verbatim}
If, after any preparation is completed, \( \text{is.good()} \) is true, \( \text{ok} \_ \neq \text{false} \) otherwise, \( \text{ok} \_ = \text{false} \). During preparation, the constructor may call \( \text{setState} (\text{failbit}) \) (which may throw \( \text{ios} \_\_\_\_\_\_\text{failure} (30.5.5.4) \))

\(~\text{sentry}();\)

\textbf{Effects: None.}

\textbf{explicit operator bool() const;}

\textbf{Effects: Returns ok\_.}

\begin{section}{Formatted input functions}

\begin{subsection}{Common requirements}

Each formatted input function begins execution by constructing an object of class \texttt{sentry} with the \texttt{noskipws} (second) argument \texttt{false}. If the \texttt{sentry} object returns \texttt{true}, when converted to a value of type \texttt{bool}, the function endeavors to obtain the requested input. If an exception is thrown during input then \( \text{ios}::\text{badbit} \) is turned on in \texttt{*this}'s error state. If \( (\text{exceptions()} \& \text{badbit}) \neq 0 \) then the exception is rethrown. In any case, the formatted input function destroys the \texttt{sentry} object. If no exception has been thrown, it returns \texttt{*this}.

\begin{subsection}{Arithmetic extractors}

\begin{verbatim}
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
\end{verbatim}

As in the case of the inserters, these extractors depend on the locale's \texttt{num-get<>} (25.4.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 30.7.4.2.1). After a \texttt{sentry} object is constructed, the conversion occurs as if performed by the following code fragment:

\begin{verbatim}
using numget = num_get<charT, istreambuf_iterator<charT, traits>>;
iosstate err = iosstate::goodbit;
use_facet<numget>(loc).get(*this, 0, *this, err, val);
setState(err);
\end{verbatim}

In the above fragment, \texttt{loc} stands for the private member of the \texttt{basic_ios} class. [\textit{Note:} The first argument provides an object of the \texttt{istreambuf_iterator} class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — end note] Class \texttt{locale} relies on this type as its interface to \texttt{istream}, so that it does not need to depend directly on \texttt{istream}.

\begin{verbatim}
operator>>(short& val);
\end{verbatim}

The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

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

\end{subsection}

\end{subsection}

\end{section}

\footnote{The \texttt{sentry} constructor and destructor can also perform additional implementation-dependent operations.}

\footnote{This is done without causing an \( \text{ios}::\text{failure} \) to be thrown.}

§ 30.7.4.2.2 1064
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);

30.7.4.2.3 basic_istream::operator>>
[istream.extractors]

basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&&));

1 Effects: None. This extractor does not behave as a formatted input function (as described in 30.7.4.2.1).
2 Returns: pf(*this).311

basic_istream<charT, traits>&
operator>>(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&&));

3 Effects: Calls pf(*this). This extractor does not behave as a formatted input function (as described in 30.7.4.2.1).
4 Returns: *this.

basic_istream<charT, traits>& operator>>(ios_base& (*pf)(ios_base&));

5 Effects: Calls pf(*this). This extractor does not behave as a formatted input function (as described in 30.7.4.2.1).
6 Returns: *this.

template<class charT, class traits>

basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT* s);
template<class traits>

basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, unsigned char* s);
template<class traits>

basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, signed char* s);

7 Effects: Behaves like a formatted input member (as described in 30.7.4.2.1) of in. After a sentry
object is constructed, operator>> extracts characters and stores them into successive locations of an
array whose first element is designated by s. If width() is greater than zero, n is width(). Otherwise
n is the number of elements of the largest array of char_type that can store a terminating charT(). n
is the maximum number of characters stored.

Characters are extracted and stored until any of the following occurs:

(8.1) n-1 characters are stored;
(8.2) end of file occurs on the input sequence;
(8.3) letting ct be use_facet<ctype<charT>>(in.getloc()), ct.is(ct.space, c) is true.

311) See, for example, the function signature ws(basic_istream&)(30.7.4.4).
312) See, for example, the function signature dec(ios_base&)(30.5.6.3).
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 (30.5.5.4).

Returns: in.

template<class charT, class traits>
    basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>& in, charT& c);

template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, unsigned char& c);

template<class traits>
    basic_istream<char, traits>& operator>>(basic_istream<char, traits>& in, signed char& c);

Effects: Behaves like a formatted input member (as described in 30.7.4.2.1) 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<char, traits>& operator>>(basic_streambuf<charT, traits>* sb);

Effects: Behaves as an unformatted input function (30.7.4.3). If sb is null, calls setstate(failbit), which may throw ios_base::failure (30.5.5.4). 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 extracts no characters, it calls setstate(failbit), which may throw ios_base::failure (30.5.5.4). If it inserted no characters because it caught an exception thrown while extracting characters from *this and failbit is on in exceptions() (30.5.5.4), then the caught exception is rethrown.

Returns: *this.

30.7.4.3 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.

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.

This is done without causing an ios::failure to be thrown.
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` (30.5.5.4).

Returns: \( c \) if available, otherwise `traits::eof()`.

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` (30.5.5.4)).

Returns: \(*this\).

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:

1. \( n \) is less than one or \( n - 1 \) characters are stored;
2. end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);
3. `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` (30.5.5.4)). In any case, if \( n \) is greater than zero it then stores a null character into the next successive location of the array.

Returns: \(*this\).

basic_istream<charT, traits>& get(char_type* s, streamsize n);

Effects: Calls `get(s, n, widen(\'\n\'))`.

Returns: Value returned by the call.

basic_istream<charT, 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` (30.5.5.4).

Returns: \(*this\).

basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb);

Effects: Calls `get(sb, widen(\'\n\'))`.

Returns: Value returned by the call.

basic_istream<charT, traits>& getline(char_type* s, streamsize n, char_type delim);

Effects: Behaves as an unformatted input function (as described above). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by \( s \). 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)`);

---

314) Note that this function is not overloaded on types `signed char` and `unsigned char`.
315) Note that this function is not overloaded on types `signed char` and `unsigned char`.
316) Note that this function is not overloaded on types `signed char` and `unsigned char`. 
2. \texttt{traits::eq(c, delim)} for the next available input character \(c\) (in which case the input character is extracted but not stored);\(^{317}\)

3. \(n\) is less than one or \(n - 1\) characters are stored (in which case the function calls \texttt{setstate(failbit)}).

These conditions are tested in the order shown.\(^ {318}\)

If the function extracts no characters, it calls \texttt{setstate(failbit)} (which may throw \texttt{ios_base::failure (30.5.5.4))}.\(^ {319}\)

In any case, if \(n\) is greater than zero, it then stores a null character (using \texttt{charT()}) into the next successive location of the array.

\textit{Returns:} *this.

[\textbf{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
            cout << "Line " << ++line_number;
        }
        cout << "(" << count << " chars): " << buffer << endl;
    }
}
```

\texttt{basic_istream<charT, traits>& getline(char_type* s, streamsize n);} \(^ {324}\)

\texttt{Returns:} getline(s, n, widen(\textquoteleft\textbackslash n\textquoteright))

\texttt{basic_istream<charT, traits>& ignore(streamsize n = 1, int_type delim = traits::eof());} \(^ {325}\)

\textit{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:

\begin{enumerate}
  \item \(n \neq \text{numeric_limits<streamsize>::max()}\) (21.3.4) and \(n\) characters have been extracted so far
  \item end-of-file occurs on the input sequence (in which case the function calls \texttt{setstate(eofbit)}, which may throw \texttt{ios_base::failure (30.5.5.4)});
  \item \texttt{traits::eq_int_type(traits::to_int_type(c), delim)} for the next available input character \(c\) (in which case \(c\) is extracted).
\end{enumerate}

\textit{Remarks:} The last condition will never occur if \texttt{traits::eq_int_type(delim, traits::eof())}.

\textit{Returns:} *this.

\(^ {317}\) Since the final input character is “extracted”, it is counted in the \texttt{gcount()}, even though it is not stored.

\(^ {318}\) This allows an input line which exactly fills the buffer, without setting \texttt{failbit}. This is different behavior than the historical AT&T implementation.

\(^ {319}\) This implies an empty input line will not cause \texttt{failbit} to be set.
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 (30.5.5.4)).
   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
   (30.5.5.4)), 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(). If rdbuf() is null, or if sputbackc() returns
   traits::eof(), calls setstate(badbit) (which may throw ios_base::failure (30.5.5.4)). [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 (30.5.5.4)). [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 (30.5.5.4), and returns -1. Otherwise, returns zero.

320) Note that this function is not overloaded on types signed char and unsigned char.

§ 30.7.4.3
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.

30.7.4.4 Standard basic_istream manipulators

```cpp
template<class charT, class traits>
basic_istream<charT, traits>& ws(basic_istream<charT, traits>& is);
```

Effects: Behaves as an unformatted input function (30.7.4.3), 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 (30.7.4.1.3). If ws stops extracting characters because there are no more available it sets eofbit, but not failbit.

Returns: is.

30.7.4.5 Rvalue stream extraction

```cpp
template<class charT, class traits, class T>
basic_istream<charT, traits>& operator>>(basic_istream<charT, traits>&& is, T&& x);
```

Effects: Equivalent to:

```cpp
is >> std::forward<T>(x);
return is;
```

Remarks: This function shall not participate in overload resolution unless the expression is >> std::forward<T>(x) is well-formed.

30.7.4.6 Class template basic_iostream

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_iostream {
        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;
    }
}
```
// 30.7.4.6.1, constructor
explicit basic_iostream(basic_streambuf<charT, traits>* sb);

// 30.7.4.6.2, destructor
virtual ~basic_iostream();

protected:
// 30.7.4.6.1, constructor
basic_iostream(const basic_iostream& rhs) = delete;
basic_iostream(basic_iostream&& rhs);

// 30.7.4.6.3, assign and swap
basic_iostream& operator=(const basic_iostream& rhs) = delete;
basic_iostream& operator=(basic_iostream&& rhs);
void swap(basic_iostream& rhs);
};

The class template basic_iostream inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

30.7.4.6.1 basic_iostream constructors

explicit basic_iostream(basic_streambuf<charT, traits>* sb);

Effects: Constructs an object of class basic_iostream, initializing the base class subobjects with basic_istream<charT, traits>(sb) (30.7.4.1) and basic_ostream<charT, traits>(sb) (30.7.5.1).

Postconditions: rdbuf() == sb and gcount() == 0.

basic_iostream(basic_iostream&& rhs);

Effects: Move constructs from the rvalue rhs by constructing the basic_istream base class with move(rhs).

30.7.4.6.2 basic_iostream destructor

virtual ~basic_iostream();

Effects: Destroys an object of class basic_iostream.

Remarks: Does not perform any operations on rdbuf().

30.7.4.6.3 basic_iostream assign 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).

30.7.5 Output streams

The header <ostream> defines a type and several function signatures that control output to a stream buffer along with a function template that inserts into stream rvalues.

30.7.5.1 Class template basic_ostream

namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ostream : virtual public basic_ios<charT, traits> {
        public:
            // types (inherited from basic_ios (30.5.5))
            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;

§ 30.7.5.1
// 30.7.5.1, constructor/destructor
explicit basic_ostream(basic_streambuf<char_type, traits>* sb);
virtual ~basic_ostream();

// 30.7.5.1.3, prefix/suffix
class sentry;

// 30.7.5.2, formatted output
basic_ostream<
    charT, traits>&
    operator<<((basic_ostream<
        charT, traits>& (*pf)(basic_ostream<
            charT, traits>&&)));

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);

// 30.7.5.3, 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();

// 30.7.5.1.4, seeks
pos_type tellp();

basic_ostream<
    charT, traits>& seekp(pos_type);

basic_ostream<
    charT, traits>& seekp(off_type, ios_base::seekdir);

protected:
// 30.7.5.1.1, copy/move constructor
basic_ostream(const basic_ostream& rhs) = delete;

basic_ostream(basic_ostream&& rhs);

// 30.7.5.1.2, assign and swap
basic_ostream& operator=(const basic_ostream& rhs) = delete;

basic_ostream& operator=(basic_ostream&& rhs);

void swap(basic_ostream& rhs);
};

// 30.7.5.2.4, character inserters
template<class charT, class traits>
    basic_ostream<
        charT, traits>& operator<<(basic_ostream<
            charT, traits>&, charT);

template<class charT, class traits>
    basic_ostream<
        charT, traits>& operator<<(basic_ostream<
            charT, traits>&, char);

template<class traits>
    basic_ostream<
        char, traits>& operator<<(basic_ostream<
            char, traits>&, signed char);
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)`.

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.

### 30.7.5.1.1 `basic_ostream` constructors

```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)` (30.5.5.2).

**Postconditions:** `rdbuf() == sb`.

```cpp
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.

```cpp
virtual ~basic_ostream();
```

**Effects:** Destroys an object of class `basic_ostream`.

**Remarks:** Does not perform any operations on `rdbuf()`.

### 30.7.5.1.2 Class `basic_ostream` assign and swap

```cpp
basic_ostream& operator=(basic_ostream&& rhs);  
```

**Effects:** As if by `swap(rhs)`.

**Returns:** `*this`.

```cpp
void swap(basic_ostream& rhs);
```

**Effects:** Calls `basic_ios<charT, traits>::swap(rhs)`.

### 30.7.5.1.3 Class `basic_ostream::sentry`

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>>
    class basic_ostream<charT, traits>::sentry {
        bool ok_;  // exposition only

        public:
            explicit sentry(basic_ostream<charT, traits>* os);
            ~sentry();
            explicit operator bool() const { return ok_; }
    }
}  // namespace std
```
The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```cpp
explicit sentry(basic_ostream<charT, traits>& os);
```

If `os.good()` is nonzero, prepares for formatted or unformatted output. If `os.tie()` is not a null pointer, calls `os.tie()->flush()`.

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` (30.5.5.4)).

```cpp
~sentry();
```

If (`os.flags() & ios_base::unitbuf`) && `!uncaught_exceptions()` && `os.good()` is true, calls `os.rdbuf()->pubsync()`. If that function returns -1, sets `badbit` in `os.rdstate()` without propagating an exception.

```cpp
explicit operator bool() const;
```

Effects: Returns `ok_`.

### 30.7.5.1.4 `basic_ostream` seek members [ostream.seeks]

Each seek member function begins execution by constructing an object of class `sentry`. It returns by destroying the `sentry` object.

```cpp
pos_type tellp();
```

Returns: If `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

```cpp
basic_ostream<charT, traits>& seekp(pos_type pos);
```

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`.

```cpp
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`.

### 30.7.5.2 Formatted output functions [ostream.formatted]

#### 30.7.5.2.1 Common requirements [ostream.formatted.reqmts]

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::failure)`, 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

---

321) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.

322) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.

323) without causing an `ios::failure` to be thrown.
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.

30.7.5.2.2 Arithmetic inserter

operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(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 imbedded locale value to perform numeric formatting. When val is of
the 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 ostream 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.

30.7.5.2.3 basic_ostream::operator<<

```cpp
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&&));
```

Effects: None. Does not behave as a formatted output function (as described in 30.7.5.2.1).

Returns: pf(*this).

```cpp
basic_ostream<charT, traits>&
operator<<(basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&&));
```

Effects: Calls pf(*this). This inserter does not behave as a formatted output function (as described in 30.7.5.2.1).

Returns: *this.

```cpp
basic_ostream<charT, traits>& operator<<(basic_streambuf<charT, traits>* sb);
```

Effects: Behaves as an unformatted output function (30.7.5.3). After the sentry object is constructed, if sb is null calls setstate(badbit) (which may throw ios_base::failure).

Gets characters from sb and inserts them in *this. Characters are read from sb and inserted until any of the following occurs:

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- an exception occurs while getting a character from sb.

If the function inserts no characters, it calls setstate(failbit) (which may throw ios_base::failure (30.5.5.4)). 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.

Returns: *this.

```cpp
basic_ostream<charT, traits>& operator<<(nullptr_t);
```

Effects: Equivalent to:

```cpp
return *this << s;
```

where s is an implementation-defined NTCTS (20.3.16).

30.7.5.2.4 Character inserter function templates

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, charT c);
```

```cpp
template<class charT, class traits>
basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, char c);
```

// specialization

```cpp
template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, char c);
```

// signed and unsigned

```cpp
template<class traits>
basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, signed char c);
```

324) See, for example, the function signature endl(basic_ostream&) (30.7.5.4).

325) See, for example, the function signature dec(ios_base&) (30.5.6.3).
template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, unsigned char c);

  Effects: Behaves as a formatted output function (30.7.5.2.1) 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 30.7.5.2.1. Inserts seq into out. Calls os.width(0).

  Returns: out.

template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const charT* s);

  template<class charT, class traits>
  basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>& out, const char* s);

  template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const char* s);

  template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const signed char* s);

  template<class traits>
  basic_ostream<char, traits>& operator<<(basic_ostream<char, traits>& out, const unsigned char* s);

  Requires: s shall not be a null pointer.

  Effects: Behaves like a formatted inserter (as described in 30.7.5.2.1) of out. Creates a character sequence seq of n characters starting at s, each widened using out.widen() (30.5.5.3), 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*;

  (4.2) char_traits<char>::length(s) for the overload where the first argument is of type basic_ostream<charT, traits>& and the second is of type const char*;

  (4.3) traits::length(reinterpret_cast<const char*>(s)) for the other two overloads.

  Determines padding for seq as described in 30.7.5.2.1. Inserts seq into out. Calls width(0).

  Returns: out.

§ 30.7.5.3 Unformatted output functions

1 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 in *this's error state. If (exceptions() & badbit) != 0 then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the unformatted output function.

basic_ostream<charT, traits>& put(char_type c);

  Effects: Behaves as an unformatted output function (as described above). After constructing a sentry object, inserts the character c, if possible.

  Otherwise, calls setstate(badbit) (which may throw ios_base::failure (30.5.5.4)).

  Returns: *this.

basic_ostream& write(const char_type* s, streamsize n);

  Effects: Behaves as an unformatted output function (as described above). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by s. Characters are inserted until either of the following occurs:

  (5.1) n characters are inserted;

---

326) without causing an ios::failure to be thrown.
327) Note that this function is not overloaded on types signed char and unsigned char.
328) Note that this function is not overloaded on types signed char and unsigned char.
— inserting in the output sequence fails (in which case the function calls \texttt{setstate(badbit)}, which may throw \texttt{ios\_base::failure} (30.5.5.4)).

\begin{verbatim}
6    Returns: *this.

basic_ostream& flush();
7    Effects: Behaves as an unformatted output function (as described above). If \texttt{rdbuf()} is not a null pointer, constructs a sentry object. If this object returns \texttt{true} when converted to a value of type \texttt{bool} the function calls \texttt{rdbuf() \rightarrow pubsync()}. If that function returns -1 calls \texttt{setstate(badbit)} (which may throw \texttt{ios\_base::failure} (30.5.5.4)). Otherwise, if the sentry object returns \texttt{false}, does nothing.
8    Returns: *this.
\end{verbatim}

30.7.5.4 Standard basic\_ostream manipulators

\begin{verbatim}
template<class charT, class traits>
    basic_ostream<charT, traits>& endl(basic_ostream<charT, traits>& os);
1    Effects: Calls \texttt{os.put(os.widen(\texttt{\'n\'}))}, then \texttt{os.flush()}.  
2    Returns: os.

template<class charT, class traits>
    basic_ostream<charT, traits>& ends(basic_ostream<charT, traits>& os);
3    Effects: Inserts a null character into the output sequence: calls \texttt{os.put(charT())}.  
4    Returns: os.

template<class charT, class traits>
    basic_ostream<charT, traits>& flush(basic_ostream<charT, traits>& os);
5    Effects: Calls \texttt{os.flush()}.  
6    Returns: os.
\end{verbatim}

30.7.5.5 Rvalue stream insertion

\begin{verbatim}
template<class charT, class traits, class T>
    basic_ostream<charT, traits>& operator<<(basic_ostream<charT, traits>&& os, const T& x);
1    Effects: As if by: \texttt{os << x};  
2    Returns: os.
3    Remarks: This function shall not participate in overload resolution unless the expression \texttt{os << x} is well-formed.
\end{verbatim}

30.7.6 Standard manipulators

The header \texttt{iomanip} defines several functions that support extractors and inserters that alter information maintained by class \texttt{ios\_base} and its derived classes.

\begin{verbatim}
unspecified resetiosflags(ios_base::fmtflags mask);
2    Returns: An object of unspecified type such that if \texttt{out} is an object of type \texttt{basic\_ostream<\texttt{charT}, traits> then the expression \texttt{out \ll resetiosflags(mask)} behaves as if it called \texttt{f(out, mask)}, or if \texttt{in} is an object of type \texttt{basic\_istream<\texttt{charT}, traits> then the expression \texttt{in \gg resetiosflags(mask)} behaves as if it called \texttt{f(in, mask)}, where the function \texttt{f} is defined as: \texttt{329}

void f(ios_base& str, ios_base::fmtflags mask) {
    // reset specified flags
    str.setf(ios_base::fmtflags(0), mask);
}

The expression \texttt{out \ll resetiosflags(mask)} shall have type \texttt{basic\_ostream<\texttt{charT}, traits>\&} and value out. The expression \texttt{in \gg resetiosflags(mask)} shall have type \texttt{basic\_istream<\texttt{charT}, traits>\&} and value in.
\end{verbatim}

\texttt{329} The expression \texttt{cin \gg resetiosflags(ios\_base::skipws)} clears \texttt{ios\_base::skipws} in the format flags stored in the \texttt{basic\_istream<\texttt{charT}, traits>} object \texttt{cin} (the same as \texttt{cin \gg noskipws}), and the expression \texttt{cout \ll resetiosflags(ios\_base::showbase)} clears \texttt{ios\_base::showbase} in the format flags stored in the \texttt{basic\_ostream<\texttt{charT}, traits>} object \texttt{cout} (the same as \texttt{cout \ll noshowbase}).
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 object of type basic_ostream<charT, traits> then the expression out << setw(n) behaves as if it called f(out, n), or if in is an object of type basic_istream<charT, traits> then the expression in >> setw(n) behaves as if it called f(in, n), where the function f is defined as:

```cpp
void f(ios_base& str, int n) {
    // set precision
    str.setf(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.
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.

30.7.7 Extended manipulators [ext.manip]

The header <iomanip> defines several functions that support extractors and inserters that allow for the
parsing and formatting of sequences and values for money and time.

```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 24).

**Effects:** The expression in >> get_money(mon, intl) described below behaves as a formatted input
function (30.7.4.2.1).

**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 24).

**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
(30.7.5.2.1) 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 MoneyPut& mp = use_facet<MoneyPut>(str.getloc());
    const Iter end = mp.put(Iter(str.rdbuf()), Intl, str, str.fill(), mon);
    if (end.failed())
        str.setstate(ios::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_istream<charT, traits>& str, struct tm* tmb, const charT* fmt) {
    using Iter = istreambuf_iterator<charT, traits>;
    using TimeGet = time_get<charT, Iter>;
    ios_base::iostate err = ios_base::goodbit;
    const TimeGet& tg = use_facet<TimeGet>(str.getloc());
    tg.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
           fmt, fmt + char_traits<charT>::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_ostream<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
    using Iter = ostreambuf_iterator<charT, traits>;
    using TimePut = time_put<charT, Iter>;
    const TimePut& tp = use_facet<TimePut>(str.getloc());
    const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
                          fmt, fmt + char_traits<charT>::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.

### 30.7.8 Quoted manipulators

[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('\\'));

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 (30.7.5.2.1) 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 30.7.5.2.1, 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('\\'));

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&) (30.7.4.2.3) which may throw ios_base::failure (30.5.3.1.1):

(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.

30.8 String-based streams

30.8.1 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>;
}
The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 24.3.

### 30.8.2 Class template `basic_stringbuf`

```cpp
namespace std {
    template<class charT, class traits = char_traits<charT>,
             class Allocator = allocator<charT>>
    class basic_stringbuf : public basic_streambuf<charT, traits> {
public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;
    using allocator_type = Allocator;

    // 30.8.2.1, constructors
    explicit basic_stringbuf(
        ios_base::openmode which = ios_base::in | ios_base::out);
    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);

    // 30.8.2.2, assign and swap
    basic_stringbuf& operator=(const basic_stringbuf& rhs) = delete;
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf& rhs);

    // 30.8.2.3, get and set
    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);

    protected:
    // 30.8.2.4, 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;
}
```

[1] The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 24.3.
pos_type seekpos(pos_type sp,
    ios_base::openmode which
    = ios_base::in | ios_base::out) override;

private:
    ios_base::openmode mode;  // exposition only
};

template<class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
    basic_stringbuf<charT, traits, Allocator>& y);

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:

— io_base::openmode mode, has in set if the input sequence can be read, and out set if the output sequence can be written.

30.8.2.1 basic_stringbuf constructors

explicit basic_stringbuf(
    io_base::openmode which = io_base::in | io_base::out);

Effects: Constructs an object of class basic_stringbuf, initializing the base class with basic_streambuf() (30.6.3.1), and initializing mode with which.

Postconditions: str() == "".

explicit basic_stringbuf(
    const basic_string<charT, traits, Allocator>& s,
    io_base::openmode which = io_base::in | io_base::out);

Effects: Constructs an object of class basic_stringbuf, initializing the base class with basic_streambuf() (30.6.3.1), and initializing mode with which. Then calls str(s).

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.

Postconditions: Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

— 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()) eptr() != rhs_a.eptr()
30.8.2.2 Assign and swap

basic_stringbuf operator=(basic_stringbuf&& rhs);

1 Effects: After the move assignment *this has the observable state it would have had if it had been move constructed from rhs (see 30.8.2.1).

2 Returns: *this.

void swap(basic_stringbuf& rhs);

3 Effects: Exchanges the state of *this and rhs.

template<class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
           basic_stringbuf<charT, traits, Allocator>& y);

4 Effects: As if by x.swap(y).

30.8.2.3 Member functions

basic_string<charT, traits, Allocator> str() const;

1 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.

void str(const basic_string<charT, traits, Allocator>& s);

2 Effects: Copies the content of s into the basic_stringbuf underlying character sequence and initializes the input and output sequences according to mode.

3 Postconditions: If mode & ios_base::out is nonzero, pbase() points to the first underlying character and egptr() >= 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.

30.8.2.4 Overridden virtual functions

int_type underflow() override;

1 Returns: If the input sequence has a read position available, returns traits::to_int_type(*gptr()). Otherwise, returns traits::eof(). Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

int_type pbackfail(int_type c = traits::eof()) override;

2 Effects: Puts back the character designated by c to the input sequence, if possible, in one of three ways:

(2.1) — If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if traits::eq(to_char_type(c), gptr()[-1]) returns true, assigns gptr() - 1 to gptr().

Returns: c.

(2.2) — If traits::eq_int_type(c, traits::eof()) returns false and if the input sequence has a putback position available, and if mode & ios_base::out is nonzero, assigns c to *--gptr().

Returns: c.

(2.3) — If traits::eq_int_type(c, traits::eof()) returns true and if the input sequence has a putback position available, assigns gptr() - 1 to gptr().

§ 30.8.2.4
Returns: `traits::not_eof(c)`.

**Returns:** As specified above, or `traits::eof()` to indicate failure.

**Remarks:** If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

```cpp
int_type overflow(int_type c = traits::eof()) override;
```

**Effects:** Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

- If `traits::eq_int_type(c, traits::eof())` returns `false` and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls `sputc(c)`. Signals success by returning `c`.

- If `traits::eq_int_type(c, traits::eof())` returns `true`, there is no character to append. Signals success by returning a value other than `traits::eof()`.

**Remarks:** The function can alter the number of write positions available as a result of any call.

**Returns:** As specified above, or `traits::eof()` to indicate failure.

The function can make a write position available only if `(mode & ios_base::out) != 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 `(mode & ios_base::in) != 0`, the function alters the read end pointer `egptr()` to point just past the new write position.

```cpp
pos_type seekoff(off_type off, ios_base::seekdir way, ios_base::openmode which = ios_base::in | ios_base::out) override;
```

**Effects:** Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 115.

### Table 115 — seekoff positioning

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(which &amp; ios_base::in) == ios_base::in</code></td>
<td>positions the input sequence</td>
</tr>
<tr>
<td><code>(which &amp; ios_base::out) == ios_base::out</code></td>
<td>positions the output sequence</td>
</tr>
<tr>
<td>`(which &amp; (ios_base::in</td>
<td>ios_base::out)) == (ios_base::in</td>
</tr>
<tr>
<td>Otherwise</td>
<td>the positioning operation fails.</td>
</tr>
</tbody>
</table>

For a sequence to be positioned, the function determines `newoff` as indicated in Table 116. If the sequence’s next pointer (either `gptr()` or `pptr()`) is a null pointer and `newoff` is nonzero, the positioning operation fails.

If `(newoff + off) < 0, or if newoff + off refers to an uninitialized character (30.8.2.3), 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))`. 

§ 30.8.2.4
Table 116 — newoff values

<table>
<thead>
<tr>
<th>Condition</th>
<th>newoff Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>way == ios_base::beg</td>
<td>0</td>
</tr>
<tr>
<td>way == ios_base::cur</td>
<td>the next pointer minus the begin-</td>
</tr>
<tr>
<td></td>
<td>ning pointer (xnext - xbeg).</td>
</tr>
<tr>
<td>way == ios_base::end</td>
<td>the high mark pointer minus the</td>
</tr>
<tr>
<td></td>
<td>beginning pointer (high_mark -</td>
</tr>
<tr>
<td></td>
<td>xbeg).</td>
</tr>
</tbody>
</table>

pos_type seekpos(pos_type sp,
    ios_base::openmode which
    = ios_base::in | ios_base::out) override;

Effects: Equivalent to seekoff(off_type(sp), ios_base::beg, which).

Returns: sp to indicate success, or pos_type(off_type(-1)) to indicate failure.

basic_streambuf<charT, traits>* setbuf(charT* s, streamsize n);

Effects: implementation-defined, except that setbuf(0, 0) has no effect.

Returns: this.

30.8.3 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;

            // 30.8.3.1, constructors
            explicit basic_istringstream(
                ios_base::openmode which = ios_base::in);
            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);

            // 30.8.3.2, assign and swap
            basic_istringstream& operator=(const basic_istringstream& rhs) = delete;
            basic_istringstream& operator=(basic_istringstream&& rhs);
            void swap(basic_istringstream& rhs);

            // 30.8.3.3, 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);
    }

§ 30.8.3
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:

\[
\begin{align*}
\text{— sb, the stringbuf object.}
\end{align*}
\]

### 30.8.3.1 `basic_istringstream` constructors

**explicit basic_istringstream(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>(&sb)` (30.7.4.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)` (30.8.2.1).

**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>(&sb)` (30.7.4.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)` (30.8.2.1).

**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(&sb)` is called to install the contained `basic_stringbuf`.

### 30.8.3.2 Assign and swap

**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`.

**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)`.

**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)`.

### 30.8.3.3 Member functions

**basic_stringbuf<charT, traits, Allocator>* rdbuf() const;**

*Returns:* `const_cast<basic_stringbuf<charT, traits, Allocator>*>(&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)`.

### 30.8.4 Class template `basic_ostringstream`

**namespace std {**

**template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT>>**

**class basic_ostringstream : public basic_ostream<charT, traits> {**

**public:**

using char_type = charT;

using int_type = typename traits::int_type;

using pos_type = typename traits::pos_type;

**}**

**§ 30.8.4**
using off_type = typename traits::off_type;
using traits_type = traits;
using allocator_type = Allocator;

// 30.8.4.1, constructors
explicit basic_ostringstream(
    ios_base::openmode which = ios_base::out);
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);

// 30.8.4.2, assign and swap
basic_ostringstream& operator=(const basic_ostringstream& rhs) = delete;
basic_ostringstream& operator=(basic_ostringstream&& rhs);
void swap(basic_ostringstream& rhs);

// 30.8.4.3, 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);

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:

(1.1)  sb, the stringbuf object.

30.8.4.1  basic_ostringstream constructors

explicit basic_ostringstream(  
    ios_base::openmode which = ios_base::out);

Effects: Constructs an object of class basic_ostringstream<charT, traits>, initializing the base class with basic_ostream<charT, traits>(&sb) (30.7.5.1) and initializing sb with basic_stringbuf<charT, traits, Allocator>(which | ios_base::out)) (30.8.2.1).

explicit basic_ostringstream(
    const basic_string<charT, traits, Allocator>& str,
    ios_base::openmode which = ios_base::out);

Effects: Constructs an object of class basic_ostringstream<charT, traits>, initializing the base class with basic_ostream<charT, traits>(&sb) (30.7.5.1) and initializing sb with basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)) (30.8.2.1).

basic_ostringstream(basic_ostringstream&& rhs);

Effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base class, and the contained basic_stringbuf. Next basic_ostream<charT, traits>::set_rdbuf(&sb) is called to install the contained basic_stringbuf.

30.8.4.2  Assign and swap

basic_ostringstream& operator=(basic_ostringstream&& rhs);

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);

3  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);

4  Effects: As if by x.swap(y).

30.8.4.3 Member functions

basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

1  Returns: const_cast<basic_stringbuf<charT, traits, Allocator>*(&sb).

basic_string<charT, traits, Allocator> str() const;

2  Returns: rdbuf()->str().

void str(const basic_string<charT, traits, Allocator>& s);

3  Effects: Calls rdbuf()->str(s).

30.8.5 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;

      // 30.8.5.1, constructors
      explicit basic_stringstream(
        ios_base::openmode which = ios_base::out | ios_base::in);
      explicit basic_stringstream(
        const basic_stringstream<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);

      // 30.8.5.2, assign and swap
      basic_stringstream& operator=(const basic_stringstream& rhs) = delete;
      basic_stringstream& operator=(basic_stringstream&& rhs);
      void swap(basic_stringstream& rhs);

      // 30.8.5.3, 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
    }
  }

30.8.5
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.

### 30.8.5.1 `basic_stringstream` constructors

```cpp
explicit basic_stringstream(
    ios_base::openmode which = ios_base::out | ios_base::in);
```

**Effects:** Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(&sb)` (30.7.4.6.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which)`.

```cpp
explicit basic_stringstream(
    const basic_string<charT, traits, Allocator>& str,
    ios_base::openmode which = ios_base::out | ios_base::in);
```

**Effects:** Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(&sb)` (30.7.4.6.1) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

```cpp
basic_stringstream(basic_stringstream&& rhs);
```

**Effects:** Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT, traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

### 30.8.5.2 Assign and swap

```cpp
basic_stringstream& operator=(basic_stringstream&& rhs);
```

**Effects:** Move assigns the base and members of `*this` from the base and corresponding members of `rhs`.

**Returns:** `*this`.

```cpp
void swap(basic_stringstream& rhs);
```

**Effects:** Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT, traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```cpp
template<class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>& x,
          basic_stringstream<charT, traits, Allocator>& y);
```

**Effects:** As if by `x.swap(y)`.

### 30.8.5.3 Member functions

```cpp
basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
```

**Returns:** `const_cast<basic_stringbuf<charT, traits, Allocator>*>(&sb)`

```cpp
basic_string<charT, traits, Allocator>& str() const;
```

**Returns:** `rdbuf()->str()`.

```cpp
void str(const basic_string<charT, traits, Allocator>& str);
```

**Effects:** Calls `rdbuf()->str(str)`.

### 30.9 File-based streams

#### 30.9.1 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>;
}
```
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>;

1 The header `<fstream>` defines four class templates and eight types that associate stream buffers with files and assist reading and writing files.

2 [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]

3 In this subclause, member functions taking arguments of `const filesystem::path::value_type*` are only be provided on systems where `filesystem::path::value_type` (30.11.7) 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]

30.9.2 Class template `basic_filebuf` [filebuf]

namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_filebuf : public basic_streambuf<charT, traits> {
  public:
    using char_type = charT;
    using int_type = typename traits::int_type;
    using pos_type = typename traits::pos_type;
    using off_type = typename traits::off_type;
    using traits_type = traits;
    // 30.9.2.1, constructors/destructor
    basic_filebuf();
    basic_filebuf(const basic_filebuf& rhs) = delete;
    basic_filebuf(basic_filebuf&& rhs);
    virtual ~basic_filebuf();
    // 30.9.2.2, assign and swap
    basic_filebuf& operator=(const basic_filebuf& rhs) = delete;
    basic_filebuf& operator=(basic_filebuf&& rhs);
    void swap(basic_filebuf& rhs);
    // 30.9.2.3, members
    bool is_open() const;
    basic_filebuf* open(const char* s, ios_base::openmode mode);
    basic_filebuf* open(const filesystem::path::value_type* s,
      ios_base::openmode mode); // wide systems only; see 30.9.1
    basic_filebuf* open(const string& s,
      ios_base::openmode mode);
    basic_filebuf* open(const filesystem::path& s,
      ios_base::openmode mode);
    basic_filebuf* close();
    protected:
    // 30.9.2.4, overridden virtual functions
    streamsize showmanyc() override;
  }
}
int_type underflow() override;
int_type uflow() override;
int_type pbackfail(int_type c = traits::eof()) override;
int_type overflow (int_type c = traits::eof()) override;

basic_streambuf<charT, traits>* setbuf(char_type* s,
    streamsize n) override;
pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which
    = ios_base::in | ios_base::out) override;
pos_type seekpos(pos_type sp,
    ios_base::openmode which
    = ios_base::in | ios_base::out) override;
int sync() override;
void imbue(const locale& loc) override;

};

template<class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
    basic_filebuf<charT, traits>& y);

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 30.9.2 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());

30.9.2.1 basic_filebuf constructors

basic_filebuf();

Effects: Constructs an object of class basic_filebuf<charT, traits>, initializing the base class with basic_streambuf<charT, traits>() (30.6.3.1).

Postconditions: 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.

Postconditions: Let rhs_p refer to the state of rhs just prior to this construction and let rhs_a refer to the state of rhs just after this construction.

(4.1) — is_open() == rhs_p.is_open()
(4.2) — rhs_a.is_open() == false
(4.3) — gptr() - eback() == rhs_p.gptr() - rhs_p.eback()
(4.4) — egptr() - eback() == rhs_p.egptr() - rhs_p.eback()
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 20.5.5.12).

30.9.2.2 Assign and swap [filebuf.assign]

basic_filebuf& operator=(basic_filebuf&& rhs);

Effects: Calls close() then move assigns from rhs. After the move assignment *this has the observable state it would have had if it had been move constructed from rhs (see 30.9.2.1).

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).

30.9.2.3 Member functions [filebuf.members]

bool is_open() const;

Returns: true if a previous call to open succeeded (returned a non-null value) and there has been no intervening call to close.

basic_filebuf* open(const char* s, ios_base::openmode mode);
basic_filebuf* open(const filesystem::path::value_type* s,
                    ios_base::openmode mode); // wide systems only; see 30.9.1

Effects: If is_open() != false, returns a null pointer. Otherwise, initializes the filebuf as required. It then opens a file, if possible, whose name is the NTBS s (as if by calling fopen(s, modstr)). The NTBS modstr is determined from mode & ~ios_base::ate as indicated in Table 117. 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)).

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);

Open(s.c_str(), mode);

330) The macro SEEK_END is defined, and the function signatures fopen(const char*, const char*) and fseek(FILE*, long, int) are declared in <cstdio> (30.12.1).
Table 117 — File open modes

<table>
<thead>
<tr>
<th>Flag combination</th>
<th>stdio equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary +</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>in +</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>out +</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>trunc +</td>
<td>&quot;wb&quot;</td>
</tr>
<tr>
<td>app +</td>
<td>&quot;wb&quot;</td>
</tr>
</tbody>
</table>

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.

Postconditions: is_open() == false.

30.9.2.4 Overridden virtual functions

streamsize showmanyc() override;

Effects: Behaves the same as basic_streambuf::showmanyc() (30.6.3.4).

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:

```cpp
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:

charT* b = pbase();
charT* p = pptr();
charT* end;
char xbuf[XSIZE];
char* xbuf_end;
codecvt_base::result r =
    a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);

and then

(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.
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 118.

<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 (25.4.1.4.2) 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.
30.9.3 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;

            // 30.9.3.1, 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 30.9.1
            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);

            // 30.9.3.2, assign and swap
            basic_ifstream& operator=(const basic_ifstream& rhs) = delete;
            basic_ifstream& operator=(basic_ifstream&& rhs);
            void swap(basic_ifstream& rhs);

            // 30.9.3.3, 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 30.9.1
            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
            };

        template<class charT, class traits>
        void swap(basic_ifstream<charT, traits>& x,
            basic_ifstream<charT, traits>& y);
    }

    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:

    — sb, the filebuf object.

30.9.3.1 basic_ifstream constructors

    basic_ifstream();
    
    Effects: Constructs an object of class basic_ifstream<charT, traits>, initializing the base class with basic_istream<charT, traits>(&sb) (30.7.4.1.1) and initializing sb with basic_filebuf<charT, traits>() (30.9.2.1).
explicit basic_ifstream(const filesystem::path::value_type* s, 
   ios_base::openmode mode = ios_base::in);  // wide systems only; see 30.9.1

Effects: Constructs an object of class basic_ifstream<charT, traits>, initializing the base class with 
basic_istream<charT, traits>(&sb) (30.7.4.1.1) and initializing sb with basic_filebuf<charT, 
traits>() (30.9.2.1), then calls rdbuf()->open(s, mode | ios_base::in). If that function returns 
a null pointer, calls setstate(failbit).

effects: Move constructs from the rvalue rhs. This is accomplished by move constructing the base 
class, and the contained basic_filebuf. Next basic_ifstream<charT, traits>:::set_rdbuf(&sb) 
is called to install the contained basic_filebuf.

30.9.3.2 Assign and swap [ifstream.assign]

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).

30.9.3.3 Member functions [ifstream.members]

basic_filebuf<charT, traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT, traits>*(&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 30.9.1

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) 
(30.5.5.4).

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) (30.5.5.4).
30.9.4 Class template basic_ofstream

```cpp
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;

        // 30.9.4.1, 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 30.9.1
        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;

        // 30.9.4.2, assign and swap
        basic_ofstream& operator=(const basic_ofstream& rhs)
            = delete;
        basic_ofstream& operator=(basic_ofstream&& rhs);
        void swap(basic_ofstream& rhs);

        // 30.9.4.3, 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 30.9.1
        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
    };
}
```

The class `basic_ofstream<charT, traits>` supports writing to named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

(1.1) — `sb`, the filebuf object.

### 30.9.4.1 basic_ofstream constructors

```cpp
basic_ofstream();
```

1 Effects: Constructs an object of class `basic_ofstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(&sb) (30.7.5.1.1)` and initializing `sb` with `basic_filebuf<charT, traits>() (30.9.2.1)`.

```cpp
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 30.9.1

Effects: Constructs an object of class basic_ofstream<charT, traits>, initializing the base class with
basic_ostream<charT, traits>(&sb) (30.7.5.1.1) and initializing sb with
basic_filebuf<charT, traits>()); (30.9.2.1), 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(&sb)
is called to install the contained basic_filebuf.

30.9.4.2 Assign 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).

30.9.4.3 Member functions

basic_filebuf<charT, traits>* rdbuf() const;

Returns: const_cast<basic_filebuf<charT, traits>*>(&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 30.9.1

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)
(30.5.5.4).

void close();

Effects: Calls rdbuf()->close() and, if that function fails (returns a null pointer), calls
setstate(failbit) (which may throw ios_base::failure) (30.5.5.4).

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).
30.9.5 Class template basic_fstream

```cpp
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;
        // 30.9.5.1, 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 30.9.1
        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);
        // 30.9.5.2, assign and swap
        basic_fstream& operator=(const basic_fstream& rhs) = delete;
        basic_fstream& operator=(basic_fstream&& rhs);
        void swap(basic_fstream& rhs);
        // 30.9.5.3, 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 30.9.1
        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:

---

---
30.9.5.1 basic_fstream constructors

basic_fstream();

1 Effects: Constructs an object of class basic_fstream<charT, traits>, initializing the base class with basic_iostream<charT, traits>(&sb) (30.7.4.6.1) and initializing sb with basic_filebuf<charT, traits>().

explicit basic_fstream(const char* s,
ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_fstream(const filesystem::path::value_type* s,
ios_base::openmode mode = ios_base::in | ios_base::out);  // wide systems only; see 30.9.1

2 Effects: Constructs an object of class basic_fstream<charT, traits>, initializing the base class with basic_iostream<charT, traits>(&sb) (30.7.4.6.1) and initializing sb with basic_filebuf<charT, traits>(). Then calls rdbuf()->open(s, mode). If that function returns a null pointer, calls setstate(failbit).

explicit basic_fstream(const string& s,
ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_fstream(const filesystem::path& s,
ios_base::openmode mode = ios_base::in | ios_base::out);

3 Effects: The same as basic_fstream(s.c_str(), mode).

basic_fstream(basic_fstream&& rhs);

4 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(&sb) is called to install the contained basic_filebuf.

30.9.5.2 Assign and swap

basic_fstream& operator=(basic_fstream&& rhs);

1 Effects: Move assigns the base and members of *this from the base and corresponding members of rhs.

Returns: *this.

void swap(basic_fstream& rhs);

2 Effects: Exchanges the state of *this and rhs by calling basic_iostream<charT,traits>::swap(rhs) and sb.swap(rhs.sb).

template<class charT, class traits>
void swap(basic_fstream<charT, traits>& x,
        basic_fstream<charT, traits>& y);

3 Effects: As if by x.swap(y).

30.9.5.3 Member functions

basic_filebuf<charT, traits>* rdbuf() const;

1 Returns: const_cast<basic_filebuf<charT, traits>*>(sb).

bool is_open() const;

2 Returns: rdbuf()->is_open().

void open(const char* s,
ios_base::openmode mode = ios_base::in | ios_base::out);
void open(
    const filesystem::path::value_type* s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
// wide systems only; see 30.9.1

Effects: Calls rdbuf()-open(s, mode). If that function does not return a null pointer calls clear(),
otherwise calls setstate(failbit) (which may throw ios_base::failure) (30.5.5.4).

void open(
    const string& s,
    ios_base::openmode mode = ios_base::in | ios_base::out);

void open(
    const filesystem::path& s,
    ios_base::openmode mode = ios_base::in | ios_base::out);

Effects: Calls open(s.c_str(), mode).

void close();

Effects: Calls rdbuf()-close() and, if that function returns a null pointer, calls setstate(failbit)
(which may throw ios_base::failure) (30.5.5.4).

30.10 Synchronized output streams

30.10.1 Header <syncstream> synopsis

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>;
}

The header <syncstream> provides a mechanism to synchronize execution agents writing to the same stream.

30.10.2 Class template basic_syncbuf

30.10.2.1 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>;

        // 30.10.2.2, 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();

        // 30.10.2.3, assignment and swap
        basic_syncbuf& operator=(basic_syncbuf&&);
        void swap(basic_syncbuf&);
    }
// 30.10.2.4, member functions
bool emit();
streambuf_type* get_wrapped() const noexcept;
allocator_type get_allocator() const noexcept;
void set_emit_on_sync(bool) noexcept;

protected:
// 30.10.2.5, overridden virtual functions
int sync() override;

private:
streambuf_type* wrapped; // exposition only
bool emit_on_sync{}; // exposition only
};

// 30.10.2.6, specialized algorithms
template<class charT, class traits, class Allocator>
void swap(basic_syncbuf<charT, traits, Allocator>&,
          basic_syncbuf<charT, traits, Allocator>&);

1 Class template basic_syncbuf stores character data written to it, known as the associated output, into internal buffers allocated using the object’s allocator. The associated output is transferred to the wrapped stream buffer object *wrapped when emit() is called or when the basic_syncbuf object is destroyed. Such transfers are atomic with respect to transfers by other basic_syncbuf objects with the same wrapped stream buffer object.

30.10.2.2 Construction and destruction [syncstream.syncbuf.cons]

basic_syncbuf(streambuf_type* obuf, const Allocator& allocator);

1 Effects: Constructs the basic_syncbuf object and sets wrapped to obuf.
2 Remarks: A copy of allocator is used to allocate memory for internal buffers holding the associated output.
3 Throws: Nothing unless an exception is thrown by the construction of a mutex or by memory allocation.
4 Postconditions: get_wrapped() == obuf and get_allocator() == allocator are true.

basic_syncbuf(basic_syncbuf&& other);

5 Effects: Move constructs from other (Table 23).
6 Postconditions: The value returned by this->get_wrapped() is the value returned by other.get_wrapped() prior to calling this constructor. Output stored in other prior to calling this constructor will be stored in *this afterwards. other.rdbuf()->pbase() == other.rdbuf()->pptr() and other.get_wrapped() == nullptr are true.
7 Remarks: This constructor disassociates other from its wrapped stream buffer, ensuring destruction of other produces no output.

~basic_syncbuf();

8 Effects: Calls emit().
9 Throws: Nothing. If an exception is thrown from emit(), the destructor catches and ignores that exception.

30.10.2.3 Assignment and swap [syncstream.syncbuf.assign]

basic_syncbuf& operator=(basic_syncbuf&& rhs) noexcept;

1 Effects: Calls emit() then move assigns from rhs. After the move assignment *this has the observable state it would have had if it had been move constructed from rhs (30.10.2.2).
2 Returns: *this.
3 Postconditions:

(3.1) rhs.get_wrapped() == nullptr is true.
this->get Allocator() == rhs.get Allocator() is true when
allocator_traits<Allocator>::propagate on_container_move_assignment::value
is true; otherwise, the allocator is unchanged.

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.

30.10.2.4 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:
  — wrapped == nullptr is false.
  — All of the characters in the associated output were successfully transferred.
  — The call to wrapped->pubsync() (if any) succeeded.

Postconditions: On success, the associated output is empty.

Synchronization: All emit() calls transferring characters to the same stream buffer object appear to
execute in a total order consistent with the “happens before” relation (6.8.2.1), 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.

30.10.2.5 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.

30.10.2.6 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).

30.10.3 Class template basic_osyncstream

30.10.3.1 Overview
	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>;

  // 30.10.3.2, construction and destruction
  basic_osyncstream(streambuf_type*, const Allocator&);
  explicit basic_osyncstream(streambuf_type* obuf)
    : basic_osyncstream(obuf, Allocator()) {} 
  basic_osyncstream(basic_ostream<charT, traits>& os, const Allocator& allocator)
    : basic_osyncstream(os.rdbuf(), allocator) {} 
  explicit basic_osyncstream(basic_ostream<charT, traits>& os)
    : basic_osyncstream(os, Allocator()) {} 
  basic_osyncstream(basic_osyncstream&&) noexcept;
  ~basic_osyncstream();

  // 30.10.3.3, assignment
  basic_osyncstream& operator=(basic_osyncstream&&) noexcept;

  // 30.10.3.4, member functions
  void emit();
  streambuf_type* get_wrapped() const noexcept;
  syncbuf_type* rdbuf() const noexcept { return &sb; }

private:
  syncbuf_type sb; // exposition only
};

1 Allocator shall meet the allocator requirements (20.5.3.5).
2 [Example: A named variable can be used within a block statement for streaming.]
   {
      osyncstream bout(cout);
      bout << "Hello, ";
      bout << "World!"
      bout << endl; // flush is noted
      bout << "and more!\n";
   } // characters are transferred and cout is flushed
   — end example]
3 [Example: A temporary object can be used for streaming within a single statement.
   osyncstream(cout) << "Hello, " << "World!" << 'n';
   In this example, cout is not flushed. — end example]

30.10.3.2 Construction and destruction [syncstream.osyncstream.cons]

basic_osyncstream(streambuf_type* buf, const Allocator& allocator);

1 Effects: Initializes sb from buf and allocator. Initializes the base class with basic_ostream(&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 Postconditions: get_wrapped() == buf is true.
basic_osyncstream(basic_osyncstream&& other) noexcept;

Effects: Move constructs the base class and sb from the corresponding subobjects of other, and calls basic_ostream<charT, traits>::set_rdbuf(&sb).

Postconditions: The value returned by get_wrapped() is the value returned by os.get_wrapped() prior to calling this constructor. nullptr == other.get_wrapped() is true.

~basic_osyncstream();

Effects: Calls emit(). If an exception is thrown from emit(), that exception is caught and ignored.

30.10.3.3 Assignment

basic_osyncstream& operator=(basic_osyncstream&& rhs) noexcept;

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]

Postconditions: nullptr == rhs.get_wrapped() is true. get_wrapped() returns the value previously returned by rhs.get_wrapped().

30.10.3.4 Member functions

void emit();

Effects: Calls sb.emit(). If that call returns false, calls setstate(ios::badbit).

[Example: A flush on a basic_osyncstream does not flush immediately:

```cpp
{ osyncstream bout(cout);
  bout << "Hello," << '"' << '\n'; // no flush
  bout.emit(); // characters transferred; cout not flushed
  bout << "World!" << endl; // flush noted; cout not flushed
  bout.emit(); // characters transferred; cout flushed
  bout << "Greetings." << '"' << '\n'; // no flush
} // characters transferred; cout not flushed
```
—end example]

[Example: The function emit() can be used to handle exceptions from operations on the underlying stream.

```cpp
{ osyncstream bout(cout);
  bout << "Hello, " << "World!" << '"' << '\n';
  try {
    bout.emit();
  } catch (...) {
    // handle exception
  }
}
```
—end example]

streambuf_type* get_wrapped() const noexcept;

Returns: sb.get_wrapped().

[Example: Obtaining the wrapped stream buffer with get_wrapped() allows wrapping it again with an osyncstream. For example,

```cpp
{ osyncstream bout1(cout);
  bout1 << "Hello, ";
  { osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '"' << '\n';
   }
  bout1 << "World!" << '"' << '\n';
}
```
produces the uninterleaved output

Goodbye, Planet!
Hello, World!

— end example]

30.11  File systems  [filesystems]

30.11.1  General  [fs.general]

This subclause describes operations on file systems and their components, such as paths, regular files, and directories.

A file system is a collection of files and their attributes.

A file is an object within a file system that holds user or system data. Files can be written to, or read from, or both. A file has certain attributes, including type. File types include regular files and directories. Other types of files, such as symbolic links, may be supported by the implementation.

A directory is a file within a file system that acts as a container of directory entries that contain information about other files, possibly including other directory files. The parent directory of a directory is the directory that both contains a directory entry for the given directory and is represented by the filename dot-dot in the given directory. The parent directory of other types of files is a directory containing a directory entry for the file under discussion.

A link is an object that associates a filename with a file. Several links can associate names with the same file. A hard link is a link to an existing file. Some file systems support multiple hard links to a file. If the last hard link to a file is removed, the file itself is removed. [Note: A hard link can be thought of as a shared-ownership smart pointer to a file. — end note] A symbolic link is a type of file with the property that when the file is encountered during pathname resolution (30.11.7), a string stored by the file is used to modify the pathname resolution. [Note: Symbolic links are often called symlinks. A symbolic link can be thought of as a raw pointer to a file. If the file pointed to does not exist, the symbolic link is said to be a “dangling” symbolic link. — end note]

30.11.2  Conformance  [fs.conformance]

Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance subclauses take that into account.

30.11.2.1  POSIX conformance  [fs.conform.9945]

Some behavior is specified by reference to POSIX (30.11.3). How such behavior is actually implemented is unspecified. [Note: This constitutes an “as if” rule allowing implementations to call native operating system or other APIs. — end note]

Implementations should provide such behavior as it is defined by POSIX. Implementations shall document any behavior that differs from the behavior defined by POSIX. Implementations that do not support exact POSIX behavior should provide behavior as close to POSIX behavior as is reasonable given the limitations of actual operating systems and file systems. If an implementation cannot provide any reasonable behavior, the implementation shall report an error as specified in 30.11.6. [Note: This allows users to rely on an exception being thrown or an error code being set when an implementation cannot provide any reasonable behavior. — end note]

Implementations are not required to provide behavior that is not supported by a particular file system. [Example: The FAT file system used by some memory cards, camera memory, and floppy disks does not support hard links, symlinks, and many other features of more capable file systems, so implementations are not required to support those features on the FAT file system but instead are required to report an error as described above. — end example]

30.11.2.2  Operating system dependent behavior conformance  [fs.conform.os]

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.

It is permissible for an implementation to be dependent upon an operating system emulator rather than the actual underlying operating system.

§ 30.11.2.2

1109
30.11.2.3 File system 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]

30.11.3 Normative references

1 This subclause mentions commercially available operating systems for purposes of exposition.\footnote{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.}

30.11.4 Requirements

1 Throughout this subclause, char, wchar_t, char16_t, and char32_t are collectively called encoded character types.

2 Functions with template parameters named EcharT shall not participate in overload resolution unless EcharT is one of the encoded character types.

3 Template parameters named InputIterator shall meet the input iterator requirements (27.2.3) 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 meet the Allocator requirements (20.5.3.5).

30.11.4.1 Namespaces and headers

1 Unless otherwise specified, references to entities described in this subclause are assumed to be qualified with ::std::filesystem::.

30.11.5 Header <filesystem> synopsis

```cpp
namespace std::filesystem {
  // 30.11.7, paths
  class path;

  // 30.11.7.6, path non-member functions
  void swap(path& lhs, path& rhs) noexcept;
  size_t hash_value(const path& p) noexcept;

  bool operator==(const path& lhs, const path& rhs) noexcept;
  bool operator!=(const path& lhs, const path& rhs) noexcept;
  bool operator<(const path& lhs, const path& rhs) noexcept;
  bool operator<=(const path& lhs, const path& rhs) noexcept;
  bool operator>(const path& lhs, const path& rhs) noexcept;
  bool operator>=(const path& lhs, const path& rhs) noexcept;

  path operator/ (const path& lhs, const path& rhs);

  // 30.11.7.6.1, path inserter and extractor
  template<class charT, class traits>
  basic_ostream<charT, traits>& operator<< (basic_ostream<charT, traits>& os, const path& p);
  template<class charT, class traits>
  basic_istream<charT, traits>& operator>> (basic_istream<charT, traits>& is, path& p);
```

\[\text{§ 30.11.5} \quad 1110\]
// 30.11.7.6.2, path factory functions
template<class Source>
    path u8path(const Source& source);
template<class InputIterator>
    path u8path(InputIterator first, InputIterator last);

// 30.11.8, filesystem errors
class filesystem_error;

// 30.11.11, directory entries
class directory_entry;

// 30.11.12, directory iterators
class directory_iterator;

// 30.11.12.2, range access for directory iterators
directory_iterator begin(directory_iterator iter) noexcept;
directory_iterator end(const directory_iterator&) noexcept;

// 30.11.13, recursive directory iterators
class recursive_directory_iterator;

// 30.11.13.2, range access for recursive directory iterators
recursive_directory_iterator begin(recursive_directory_iterator iter) noexcept;
recursive_directory_iterator end(const recursive_directory_iterator&) noexcept;

// 30.11.10, file status
class file_status;

struct space_info {
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
};

// 30.11.9, 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<trivial-clock>;

// 30.11.14, 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) noexcept;
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) noexcept;

bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec) noexcept;
bool copy_file(const path& from, const path& to, copy_options options);
bool copy_file(const path& from, const path& to, copy_options option, error_code& ec) noexcept;

§ 30.11.5

1111
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) noexcept;

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) noexcept;

bool is_fifo(file_status s) noexcept;
bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;

§ 30.11.5 1112
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) noexcept;

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);
trivial-clock is an implementation-defined type that satisfies the TrivialClock requirements (23.17.3)
and that is capable of representing and measuring file time values. Implementations should ensure that the
resolution and range of file_time_type reflect the operating system dependent resolution and range of file
time values.

30.11.6 Error reporting

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.

—end note]

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 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 20.5.5.12.

(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.

30.11.7 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 (30.11.7.1). The maximum number of
elements in the sequence is operating system dependent (30.11.2.2).

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 (30.11.7.1) 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]

namespace std::filesystem {
    class path {
        public:
            using value_type = see below;
            using string_type = basic_string<value_type>;
            static constexpr value_type preferred_separator = see below;

            // 30.11.9.1, enumeration format
            enum format;

            // 30.11.7.4.1, constructors and destructor
            path() noexcept;
            path(const path& p);
            path(path&& p) noexcept;
            path(string_type&& source, format fmt = auto_format);
            template<class Source>
                path(const Source& source, format fmt = auto_format);
            template<class InputIterator>
                path(InputIterator first, InputIterator last, format fmt = auto_format);
            ~path();

            // 30.11.7.4.2, assignments
            path& operator=(const path& p);
            path& operator=(path&& p) noexcept;
            path& assign(string_type& source);
            template<class Source>
                path& operator=(const Source& source);
            template<class InputIterator>
                path& assign(InputIterator first, InputIterator last);

            // 30.11.7.4.3, appends
            path& operator/=(const path& p);
            template<class Source>
                path& operator/=(const Source& x);
            template<class InputIterator>
                path& append(InputIterator first, InputIterator last);

            // 30.11.7.4.4, 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);
            template<class Source>
                path& operator+=(const Source& x);
            template<class EcharT>
                path& operator+=(EcharT x);
            template<class InputIterator>
                path& concat(InputIterator first, InputIterator last);
        }
    }

§ 30.11.7
// 30.11.7.4.5, 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;

// 30.11.7.4.6, 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::string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;

// 30.11.7.4.7, 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::string generic_u8string() const;
std::u16string generic_u16string() const;
std::u32string generic_u32string() const;

// 30.11.7.4.8, 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;

// 30.11.7.4.9, 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;

// 30.11.7.4.10, 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;
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 (30.11.7.1).

[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]

### 30.11.7.1 Generic pathname format

pathname:

- root-name_opt root-directory_opt relative-path

- root-name:
  - operating system dependent sequences of characters
  - implementation-defined sequences of characters

- root-directory:
  - directory-separator

- relative-path:
  - filename directory-separator relative-path
  - an empty path

- filename:
  - non-empty sequence of characters other than directory-separator characters

- directory-separator:
  - preferred-separator directory-separator_opt
  - fallback-separator directory-separator_opt

- preferred-separator:
  - operating system dependent directory separator character

- fallback-separator:
  - /, if preferred-separator is not /

A filename is the name of a file. Filenames dot and dot-dot, consisting solely of one and two period characters respectively, have special meaning. The following characteristics of filenames are operating system dependent:

1. The permitted characters. [Example: Some operating systems prohibit the ASCII control characters (0x00 – 0x1F) in filenames. —end example] [Note: For wide portability, users may wish to limit filename characters to the POSIX Portable Filename Character Set: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z 0 1 2 3 4 5 6 7 8 9 . _ – — end note]

2. The maximum permitted length.

3. Filenames that are not permitted.

4. Filenames that have special meaning.

5. Case awareness and sensitivity during path resolution.

6. Special rules that may apply to file types other than regular files, such as directories.
Except in a root-name, multiple successive directory-separator characters are considered to be the same as one directory-separator character.

The filename dot is treated as a reference to the current directory. The filename dot-dot is treated as a reference to the parent directory. What the filename dot-dot 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 (30.11.7). 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 (30.11.7.1) 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.

30.11.7.2 path conversions [fs.path.cvt]

30.11.7.2.1 path argument format conversions [fs.path.fmt.cvt]

[Note: The format conversions described in this subclause are not applied on POSIX-based operating systems because on these systems:

(1.1) — The generic format is acceptable as a native path.
(1.2) — There is no need to distinguish between native format and generic format in function arguments.
(1.3) — Paths for regular files and paths for directories share the same syntax.

— end note]

Several functions are defined to accept detected-format arguments, which are character sequences. A detected-format argument represents a path using either a pathname in the generic format (30.11.7.1) or a pathname in the native format (30.11.7). Such an argument is taken to be in the generic format if and only if it matches the generic format and is not acceptable to the operating system as a native path.

[Note: Some operating systems may have no unambiguous way to distinguish between native format and generic format arguments. This is by design as it simplifies use for operating systems that do not require disambiguation. An implementation for an operating system where disambiguation is required is permitted to distinguish between the formats. — end note]

Pathnames are converted as needed between the generic and native formats in an operating-system-dependent manner. Let G(n) and N(g) in a mathematical sense be the implementation’s functions that convert native-to-generic and generic-to-native formats respectively. If g=G(n) for some n, then G(N(g))=g; if n=N(g) for some g, then N(G(n))=n. [Note: Neither G nor N need be invertible. — end note]

If the native format requires paths for regular files to be formatted differently from paths for directories, the path shall be treated as a directory path if its last element is a directory-separator, otherwise it shall be treated as a path to a regular file.
6 [Note: A path stores a native format pathname (30.11.7.4.6) 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 \((G \lor N)\).

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]

**30.11.7.2.2** path type and encoding conversions  
[fs.path.type.cvt]

1 The native encoding of a narrow character string is the operating system dependent current encoding for pathnames (30.11.7). The native encoding for wide character strings is the implementation-defined execution wide-character set encoding (5.3).

2 For member function arguments that take character sequences representing paths and for member functions returning strings, value type and encoding conversion is performed if the value type of the argument or return value differs from path::value_type. For the argument or return value, the method of conversion and the encoding to be converted to is determined by its value type:

(2.1) — char: The encoding is the native narrow 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 narrow 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 narrow character strings to identify paths. Changing this behavior would be surprising and error prone. —end note]

(2.2) — wchar_t: The encoding is the native wide encoding. The method of conversion is unspecified. [Note: For Windows-based operating systems path::value_type is wchar_t so no conversion from wchar_t value type arguments or to wchar_t value type return values is performed. —end note]

(2.3) — char16_t: The encoding is UTF-16. The method of conversion is unspecified.

(2.4) — char32_t: The encoding is UTF-32. The method of conversion is unspecified.

3 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.

**30.11.7.3** path requirements  
[fs.path.req]

1 In addition to the requirements (30.11.4), function template parameters named Source shall be one of:

(1.1) — basic_string<EcharT, traits, Allocator>. A function argument const Source& source shall have an effective range \([source.begin(), source.end())\).

(1.2) — basic_string_view<EcharT, traits>. A function argument const Source& source shall have an effective range \([source.begin(), source.end())\).

(1.3) — A type meeting the input iterator requirements that iterates over a NTCTS. The value type shall be an encoded character type. A function argument const Source& source shall have an effective range \([source, end)\) where end is the first iterator value with an element value equal to iterator_traits<Source>::value_type().

(1.4) — A character array that after array-to-pointer decay results in a pointer to the start of a NTCTS. The value type shall be an encoded character type. A function argument const Source& source shall have an effective range \([source, end)\) where end is the first iterator value with an element value equal to iterator_traits<decay_t<Source>>::value_type().

2 Functions taking template parameters named Source shall not participate in overload resolution unless either

(2.1) — Source is a specialization of basic_string or basic_string_view, or

(2.2) — the qualified-id iterator_traits<decay_t<Source>>::value_type is valid and denotes a possibly const encoded character type (17.9.2).
Arguments of type `Source` shall not be null pointers.

### 30.11.7.4 path members

#### 30.11.7.4.1 path constructors

- `path() noexcept;`
  
  **Effects:** Constructs an object of class `path`.

- `path(const path& p);`
  
  **Postconditions:** `empty() == true`.

- `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` (30.11.7.2.1), converting format if required (30.11.7.2.1). `source` is left in a valid but unspecified state.

- `template<class Source>
  path(const Source& source, format fmt = auto_format);`

- `template<class InputIterator>
  path(InputIterator first, InputIterator last, format fmt = auto_format);`
  
  **Effects:** Let `s` be the effective range of `source` (30.11.7.3) or the range `[first, last)`, with the encoding converted if required (30.11.7.2). Finds the detected-format of `s` (30.11.7.2.1) and constructs an object of class `path` for which the pathname in that format is `s`.

- `template<class Source>
  path(const Source& source, const locale& loc, format fmt = auto_format);`

- `template<class InputIterator>
  path(InputIterator first, InputIterator last, const locale& loc, format fmt = auto_format);`
  
  **Requires:** The value type of `Source` and `InputIterator` is `char`.

  **Effects:** Let `s` be the effective range of `source` or the range `[first, last)`, after converting the encoding as follows:

  1. If `value_type` is `wchar_t`, converts to the native wide encoding (30.11.7.2.2) using the `codecvt<wchar_t, char, mbstate_t>` facet of `loc`.
  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 narrow encoding.

  Finds the detected-format of `s` (30.11.7.2.1) and constructs an object of class `path` for which the pathname in that format is `s`.

[Example:] A string is to be read from a database that is encoded in ISO/IEC 8859-1, and used to create a directory:

```cpp
namespace fs = std::filesystem;
std::string latin1_string = read_latin1_data();
codecvt<8859_1<wchar_t> latin1_facet;
std::locale latin1_locale(std::locale(), latin1_facet);
fs::create_directory(fs::path(latin1_string, latin1_locale));
```

For POSIX-based operating systems, the path is constructed by first using `latin1_facet` to convert ISO/IEC 8859-1 encoded `latin1_string` to a wide character string in the native wide encoding (30.11.7.2.2). The resulting wide string is then converted to a narrow character pathname string in the current native narrow encoding. If the native wide encoding is UTF-16 or UTF-32, and the current native narrow 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 narrow encodings some characters may have no representation.
For Windows-based operating systems, the path is constructed by using `latin1_facet` to convert ISO/IEC 8859-1 encoded `latin1_string` to a UTF-16 encoded wide character pathname string. All of the characters in the ISO/IEC 8859-1 character set will be converted to their Unicode representation.

--- end example

30.11.7.4.2 `path` assignments

```cpp
path& operator=(const path& p);
```

1. **Effects:** If `*this` and `p` are the same object, has no effect. Otherwise, sets both respective pathnames of `*this` to the respective pathnames of `p`.
2. **Returns:** `*this`.

```cpp
path& operator=(path&& p) noexcept;
```

3. **Effects:** If `*this` and `p` are the same object, has no effect. Otherwise, sets both respective pathnames of `*this` to the respective pathnames of `p`. `p` is left in a valid but unspecified state.  
   
   **Note:** A valid implementation is `swap(p)`.  
   
   --- end note

4. **Returns:** `*this`.

```cpp
path& operator=(string_type&& source);
path& assign(string_type&& source);
```

5. **Effects:** Sets the pathname in the detected-format of `source` to the original value of `source`. `source` is left in a valid but unspecified state.
6. **Returns:** `*this`.

```cpp
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);
```

7. **Effects:** Let `s` be the effective range of `source` (30.11.7.3) or the range `[first, last)`, with the encoding converted if required (30.11.7.2). Finds the detected-format of `s` (30.11.7.2.1) and sets the pathname in that format to `s`.
8. **Returns:** `*this`.

30.11.7.4.3 `path` appends

The append operations use `operator//=` to denote their semantic effect of appending `preferred-separator` when needed.

```cpp
path& operator/=(const path& p);
```

1. **Effects:** If `p.is_absolute() || (p.has_root_name() && p.root_name() != root_name())`, then `operator=(p)`.  
2. Otherwise, modifies `*this` as if by these steps:
3.  
4.   3.1. — If `p.has_root_directory()`, then removes any root directory and relative path from the generic format pathname. Otherwise, if `!has_root_directory() && is_absolute()` is `true` or if `has_filename()` is `true`, then appends `path::preferred_separator` to the generic format pathname.
5.   3.2. — Then appends the native format pathname of `p`, omitting any `root-name` from its generic format pathname, to the native format pathname.
6. **Example:** Even if `//host` is interpreted as a `root-name`, both of the paths `path("//host")/"foo"` and `path("//host")/"foo"` equal `"//host/foo"`.

Expression examples:

// On POSIX,  
`path("foo") / ""`;  // yields "foo/"  
`path("foo") / "/bar";`  // yields "/bar"

// On Windows, backslashes replace slashes in the above yields

§ 30.11.7.4.3 1121
// On Windows,
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]

Returns: *this.

template<class Source>
path& operator/=(const Source& source);
template<class Source>
path& append(const Source& source);
Effects: Equivalent to: return operator/=(path(source));
template<class InputIterator>
path& append(InputIterator first, InputIterator last);
Effects: Equivalent to: return operator/=(path(first, last));

30.11.7.4.4 path 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);
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]

Returns: *this.
template<class InputIterator>
path& concat(InputIterator first, InputIterator last);
Effects: Equivalent to: return *this += path(first, last);

30.11.7.4.5 path modifiers

void clear() noexcept;
Postconditions: empty() == true.
path& make_preferred();
Effects: Each directory-separator of the pathname in the generic format is converted to preferred-separator.
Returns: *this.

[Example:
path p("foo/bar");
std::cout << p << \
'\n';
p.make_preferred();
std::cout << p << \
'\n';
On an operating system where preferred-separator is a slash, the output is:
"foo/bar"
"foo/bar"
On an operating system where preferred-separator is a backslash, the output is:
"foo/bar"
"foo\bar"

— end example]

path& remove_filename();

Postconditions: !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").replace_filename("bar"); // yields "/bar" on POSIX
  path("/").replace_filename("bar"); // yields "/bar" on POSIX

— end example]

path& replace_extension(const path& replacement = path());

Effects:
  — (12.1) Any existing extension() (30.11.7.4.9) is removed from the pathname in the generic format, then
  — (12.2) If replacement is not empty and does not begin with a dot character, a dot character is appended to the pathname in the generic format, then
  — (12.3) operator+=(replacement);

Returns: *this.

void swap(path& rhs) noexcept;

Effects: Swaps the contents (in all formats) of the two paths.

Complexity: Constant time.

30.11.7.4.6 path native format observers [fs.path.native.obs]

The string returned by all native format observers is in the native pathname format (30.11.7).

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 30.11.7.2.

std::string string() const;
std::wstring wstring() const;
std::string u8string() const;
std::u16string u16string() const;
std::u32string u32string() const;

Returns: native().

Remarks: Conversion, if any, is performed as specified by 30.11.7.2. The encoding of the string returned by u8string() is always UTF-8.

30.11.7.4.7 path generic format observers

Generic format observer functions return strings formatted according to the generic pathname format (30.11.7.1). 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 30.11.7.2.

std::string generic_string() const;
std::wstring generic_wstring() const;
std::string 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 30.11.7.2. The encoding of the string returned by generic_u8string() is always UTF-8.

30.11.7.4.8 path compare

int compare(const path& p) const noexcept;

Returns:

(1.1) A value less than 0, if native() for the elements of *this are lexicographically less than native() for the elements of p; otherwise,

(1.2) a value greater than 0, if native() for the elements of *this are lexicographically greater than native() for the elements of p; otherwise,

(1.3) 0.

Remarks: The elements are determined as if by iteration over the half-open range [begin(), end()) for *this and p.

int compare(const string_type& s) const
int compare(basic_string_view<value_type> s) const;

Returns: compare(path(s)).
int compare(const value_type* s) const  

    Returns: compare(path(s)).

30.11.7.4.9  path decomposition  
[fs.path.decompose]

path root_name() const;  

    Returns: root-name, if the pathname in the generic format includes root-name, otherwise path().

path root_directory() const;  

    Returns: root-directory, if the pathname in the generic format includes root-directory, otherwise path().

path root_path() const;  

    Returns: root_name() / root_directory().

path relative_path() const;  

    Returns: A path composed from the pathname in the generic format, if empty() is false, beginning with the first filename after root-path. Otherwise, path().

path parent_path() const;  

    Returns: *this if has_relative_path() is false, otherwise a path whose generic format pathname is the longest prefix of the generic format pathname of *this that produces one fewer element in its iteration.

path filename() const;  

    Returns: relative_path().empty() ? path() : *--end().

[Example:

path("/foo/bar.txt").filename(); // yields "bar.txt"
path("/foo/bar").filename(); // yields "bar"
path("/foo/bar/").filename(); // yields ""
path("/").filename(); // yields ""
path("//host").filename(); // yields ""
path(".").filename(); // yields "." 
path("..").filename(); // yields "."  
— end example]

path stem() const;  

    Returns: Let f be the generic format pathname of filename(). Returns a path whose pathname in the generic format is

(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;  

    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 "." 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

30.11.7.4.10 path 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 (30.11.7), 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().

30.11.7.4.11 path generation

path lexically_normal() const;

Returns: A path whose pathname in the generic format is the normal form (30.11.7.1) of the pathname in the generic format of *this.

[Example:
assert(path("foo/./bar/..").lexically_normal() == "foo/");
assert(path("foo////bar/../").lexically_normal() == "foo/");
The above assertions will succeed. On Windows, the returned path's directory-separator characters will be backslashes rather than slashes, but that does not affect path equality. — end example]
path lexically_relative(const path& base) const;

Returns: *this made relative to base. Does not resolve (30.11.7) symlinks. Does not first normalize (30.11.7.1) *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:

```
auto [a, b] = mismatch(begin(), end(), base.begin(), base.end());
```

Then,

(4.1) — if a == end() and b == base.end(), returns path("."); otherwise
(4.2) — let n be the number of filename elements in [b, base.end()) that are not dot or dot-dot minus the number that are dot-dot. If n<0, returns path(); otherwise
(4.3) — returns an object of class path that is default-constructed, followed by
   (4.3.1) — application of operator/=(path("..")) n times, and then
   (4.3.2) — application of operator/= for each element in [a, end()).

Example:

```
assert(path("/a/d").lexically_relative("/a/b/c") == "../../d");
assert(path("/a/b/c").lexically_relative("/a/d") == "/..\b/c");
assert(path("a/b/c").lexically_relative("a") == "/b/c");
assert(path("a/b/c").lexically_relative("a/b/c/x/y") == "/..\..");
assert(path("a/b/c").lexically_relative("a/b/c") == "/.");
assert(path("a/b").lexically_relative("c/d") == "/../a/b");
```

The above assertions will succeed. On Windows, the returned path’s directory-separator characters will be backslashes rather than slashes, but that does not affect path equality. — end example]

[Note: If symlink following semantics are desired, use the operational function relative(). — end note]

Note: If normalization (30.11.7.1) is needed to ensure consistent matching of elements, apply lexically_normal() to *this, base, or both. — end note]

path lexically_proximate(const path& base) const;

Returns: If the value of lexically_relative(base) is not an empty path, return it. Otherwise return *this.

[Note: If symlink following semantics are desired, use the operational function proximate(). — end note]

[Note: If normalization (30.11.7.1) is needed to ensure consistent matching of elements, apply lexically_normal() to *this, base, or both. — end note]

30.11.7.5 path iterators

Path iterators iterate over the elements of the pathname in the generic format (30.11.7.1).

A path::iterator is a constant iterator satisfying all the requirements of a bidirectional iterator (27.2.6) except that, for dereferenceable iterators a and b of type path::iterator with a == b, there is no requirement that *a and *b are bound to the same object. Its value_type is path.

Calling any non-const member function of a path object invalidates all iterators referring to elements of that object.

For the elements of the pathname in the generic format, the forward traversal order is as follows:

(4.1) — The root-name element, if present.
(4.2) — 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.

§ 30.11.7.5 1127
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.

30.11.7.6 path non-member functions

    [fs.path.nonmember]

void swap(path& lhs, path& rhs) noexcept;

    Effects: Equivalent to lhs.swap(rhs).

size_t hash_value (const path& p) noexcept;

    Returns: A hash value for the path p. If for two paths, p1 == p2 then hash_value(p1) == hash_value(p2).

bool operator< (const path& lhs, const path& rhs) noexcept;

    Returns: lhs.compare(rhs) < 0.

bool operator<=(const path& lhs, const path& rhs) noexcept;

    Returns: !(rhs < lhs).

bool operator> (const path& lhs, const path& rhs) noexcept;

    Returns: rhs < lhs.

bool operator>=(const path& lhs, const path& rhs) noexcept;

    Returns: !(lhs < rhs).

bool operator==(const path& lhs, const path& rhs) noexcept;

    Returns: !(lhs < rhs) && !(rhs < lhs).

    [Note: Path equality and path equivalence have different semantics.

    (8.1) — Equality is determined by the path non-member operator==, which considers the two path’s lexical representations only. [Example: path("foo") == "bar" is never true. — end example]

    (8.2) — Equivalence is determined by the equivalent() non-member function, which determines if two paths resolve (30.11.7) 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]

bool operator!=(const path& lhs, const path& rhs) noexcept;

    Returns: !(lhs == rhs).

path operator/ (const path& lhs, const path& rhs);

    Effects: Equivalent to: return path(lhs) /= rhs;

30.11.7.6.1 path inserter and extractor

    [fs.path.io]

template<class charT, class traits>
    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 30.7.8. — end note]

    Returns: os.
template<class charT, class traits>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, path& p);

Effects: Equivalent to:
    basic_string<charT, traits> tmp;
    is >> quoted(tmp);
    p = tmp;

Returns: is.

30.11.7.6.2 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.

Returns:

(2.1) — If value_type is char and the current native narrow encoding (30.11.7.2.2) 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 (30.11.7.2.1) applies to the arguments for these functions. How Unicode encoding conversions are performed is unspecified.

[Example: A string is to be read from a database that is encoded in UTF-8, and used to create a directory using the native encoding for filenames:

namespace fs = std::filesystem;
std::string utf8_string = read_utf8_data();
fs::create_directory(fs::u8path(utf8_string));

For POSIX-based operating systems with the native narrow encoding set to UTF-8, no encoding or type conversion occurs.

For POSIX-based operating systems with the native narrow encoding not set to UTF-8, a conversion to UTF-32 occurs, followed by a conversion to the current native narrow encoding. Some Unicode characters may have no native character set representation.

For Windows-based operating systems a conversion from UTF-8 to UTF-16 occurs. — end example]

30.11.8 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.
30.11.8.1 filesystem_error members

Constructors are provided that store zero, one, or two paths associated with an error.

```
filesystem_error(const string& what_arg, error_code ec);
```

Postconditions: The postconditions of this function are indicated in Table 119.

Table 119 — filesystem_error(const string&, error_code) effects

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime_error::what()</td>
<td>what_arg.c_str()</td>
</tr>
<tr>
<td>code()</td>
<td>ec</td>
</tr>
<tr>
<td>path1().empty()</td>
<td>true</td>
</tr>
<tr>
<td>path2().empty()</td>
<td>true</td>
</tr>
</tbody>
</table>

```
filesystem_error(const string& what_arg, const path& p1, error_code ec);
```

Postconditions: The postconditions of this function are indicated in Table 120.

Table 120 — filesystem_error(const string&, const path&, error_code) effects

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime_error::what()</td>
<td>what_arg.c_str()</td>
</tr>
<tr>
<td>code()</td>
<td>ec</td>
</tr>
<tr>
<td>path1()</td>
<td>Reference to stored copy of p1</td>
</tr>
<tr>
<td>path2().empty()</td>
<td>true</td>
</tr>
</tbody>
</table>

```
filesystem_error(const string& what_arg, const path& p1, const path& p2, error_code ec);
```

Postconditions: The postconditions of this function are indicated in Table 121.

Table 121 — filesystem_error(const string&, const path&, const path&, error_code) effects

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime_error::what()</td>
<td>what_arg.c_str()</td>
</tr>
<tr>
<td>code()</td>
<td>ec</td>
</tr>
<tr>
<td>path1()</td>
<td>Reference to stored copy of p1</td>
</tr>
<tr>
<td>path2()</td>
<td>Reference to stored copy of p2</td>
</tr>
</tbody>
</table>

```
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: A string containing runtime_error::what(). 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.

30.11.9 Enumerations

30.11.9.1 Enum path::format

This enum specifies constants used to identify the format of the character sequence, with the meanings listed in Table 122.

30.11.9.2 Enum class file_type

This enum class specifies constants used to identify file types, with the meanings listed in Table 123.
Table 122 — 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>

Table 123 — 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 <code>file_type</code> types shall supply implementation-defined <code>file_type</code></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>

30.11.9.3 Enum class `copy_options`  

The `enum` class type `copy_options` is a bitmask type (20.4.2.1.4) 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 124. Constant `none` is shown in each option group for purposes of exposition; implementations shall provide only a single definition.

30.11.9.4 Enum class `perms`  

The `enum` class type `perms` is a bitmask type (20.4.2.1.4) that specifies bitmask constants used to identify file permissions, with the meanings listed in Table 125.

30.11.9.5 Enum class `perm_options`  

The `enum` class type `perm_options` is a bitmask type (20.4.2.1.4) that specifies bitmask constants used to control the semantics of permissions operations, with the meanings listed in Table 126. The bitmask constants are bitmask elements. In Table 126 `perm` denotes a value of type `perms` passed to `permissions`.

30.11.9.6 Enum class `directory_options`  

The `enum` class type `directory_options` is a bitmask type (20.4.2.1.4) that specifies bitmask constants used to identify directory traversal options, with the meanings listed in Table 127.

30.11.10 Class `file_status`  

```cpp
namespace std::filesystem {
    class file_status {
    public:
        // 30.11.10.1, constructors and destructor
```
Table 124 — Enum class `copy_options`

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Error; file already exists.</td>
</tr>
<tr>
<td>skip_existing</td>
<td>Do not overwrite existing file, do not report an error.</td>
</tr>
<tr>
<td>overwrite_existing</td>
<td>Overwrite the existing file.</td>
</tr>
<tr>
<td>update_existing</td>
<td>Overwrite the existing file if it is older than the replacement file.</td>
</tr>
</tbody>
</table>

Option group controlling copy function effects for existing target files

Option group controlling copy function effects for sub-directories

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Do not copy sub-directories.</td>
</tr>
<tr>
<td>recursive</td>
<td>Recursively copy sub-directories and their contents.</td>
</tr>
</tbody>
</table>

Option group controlling copy function effects for symbolic links

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Follow symbolic links.</td>
</tr>
<tr>
<td>copy_symlinks</td>
<td>Copy symbolic links as symbolic links rather than copying the files that they point to.</td>
</tr>
<tr>
<td>skip_symlinks</td>
<td>Ignore symbolic links.</td>
</tr>
</tbody>
</table>

Option group controlling copy function effects for choosing the form of copying

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>(Default) Copy content.</td>
</tr>
<tr>
<td>directories_only</td>
<td>Copy directory structure only, do not copy non-directory files.</td>
</tr>
<tr>
<td>create_symlinks</td>
<td>Make symbolic links instead of copies of files. The source path shall be an absolute path unless the destination path is in the current directory.</td>
</tr>
<tr>
<td>create_hard_links</td>
<td>Make hard links instead of copies of files.</td>
</tr>
</tbody>
</table>

Table 125 — Enum class `perms`

<table>
<thead>
<tr>
<th>Name</th>
<th>Value (octal)</th>
<th>POSIX macro</th>
<th>Definition or notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>0</td>
<td>S_IRUSR</td>
<td>There are no permissions set for the file.</td>
</tr>
<tr>
<td>owner_read</td>
<td>0400</td>
<td>S_IRUSR</td>
<td>Read permission, owner</td>
</tr>
<tr>
<td>owner_write</td>
<td>0200</td>
<td>S_IWUSR</td>
<td>Write permission, owner</td>
</tr>
<tr>
<td>owner_exec</td>
<td>0100</td>
<td>S_IXUSR</td>
<td>Execute/search permission, owner</td>
</tr>
<tr>
<td>owner_all</td>
<td>0700</td>
<td>S_IRWXU</td>
<td>Read, write, execute/search by owner; owner_read</td>
</tr>
<tr>
<td>group_read</td>
<td>040</td>
<td>S_IRGRP</td>
<td>Read permission, group</td>
</tr>
<tr>
<td>group_write</td>
<td>020</td>
<td>S_IWGRP</td>
<td>Write permission, group</td>
</tr>
<tr>
<td>group_exec</td>
<td>010</td>
<td>S_IXGRP</td>
<td>Execute/search permission, group</td>
</tr>
<tr>
<td>group_all</td>
<td>070</td>
<td>S_IRWXG</td>
<td>Read, write, execute/search by group; group_read</td>
</tr>
<tr>
<td>others_read</td>
<td>04</td>
<td>S_IROTH</td>
<td>Read permission, others</td>
</tr>
<tr>
<td>others_write</td>
<td>02</td>
<td>S_IWOTH</td>
<td>Write permission, others</td>
</tr>
<tr>
<td>others_exec</td>
<td>01</td>
<td>S_IXOTH</td>
<td>Execute/search permission, others</td>
</tr>
<tr>
<td>others_all</td>
<td>07</td>
<td>S_IRWXO</td>
<td>Read, write, execute/search by others; others_read</td>
</tr>
<tr>
<td>all</td>
<td>0777</td>
<td>owner_all</td>
<td>group_all</td>
</tr>
<tr>
<td>set_uid</td>
<td>04000</td>
<td>S_ISUID</td>
<td>Set-user-ID on execution</td>
</tr>
<tr>
<td>set_gid</td>
<td>02000</td>
<td>S_ISGID</td>
<td>Set-group-ID on execution</td>
</tr>
<tr>
<td>sticky_bit</td>
<td>01000</td>
<td>S_ISVTX</td>
<td>Operating system dependent.</td>
</tr>
<tr>
<td>mask</td>
<td>07777</td>
<td>all</td>
<td>set_uid</td>
</tr>
<tr>
<td>unknown</td>
<td>0xFFFF</td>
<td>The permissions are not known, such as when a file-status object is created without specifying the permissions</td>
<td></td>
</tr>
</tbody>
</table>
Table 126 — Enum class perm_options

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>permissions shall replace the file's permission bits with perm</td>
</tr>
<tr>
<td>add</td>
<td>permissions shall replace the file's permission bits with the bitwise OR of perm and the file's current permission bits.</td>
</tr>
<tr>
<td>remove</td>
<td>permissions shall replace the file’s permission bits with the bitwise AND of the complement of perm and the file’s current permission bits.</td>
</tr>
<tr>
<td>nofollow</td>
<td>permissions shall change the permissions of a symbolic link itself rather than the permissions of the file the link resolves to.</td>
</tr>
</tbody>
</table>

Table 127 — 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() noexcept : file_status(file_type::none) {}
explicit file_status(file_type ft, 
perms prms = perms::unknown) noexcept;
file_status(const file_status&) noexcept = default;
file_status(file_status&) noexcept = default;
-file_status();

// assignments
file_status& operator=(const file_status&) noexcept = default;
file_status& operator=(file_status&&) noexcept = default;

// 30.11.10.3, modifiers
void type(file_type ft) noexcept;
void permissions(perms prms) noexcept;

// 30.11.10.2, 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.

30.11.10.1 file_status constructors

explicit file_status(file_type ft, perms prms = perms::unknown) noexcept;

1 Postconditions: type() == ft and permissions() == prms.

30.11.10.2 file_status observers

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.
30.11.10.3 file_status modifiers

void type(file_type ft) noexcept;

Postconditions: type() == ft.

void permissions(perms prms) noexcept;

Postconditions: permissions() == prms.

30.11.11 Class directory_entry

namespace std::filesystem {

class directory_entry {

public:

  // 30.11.11.1, 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;

  // 30.11.11.2, 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;

  // 30.11.11.3, 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(const filesystem::path& 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;
  bool is_socket() const;
  bool is_socket(const filesystem::path& ec) const noexcept;
  bool is_symlink() const;
  bool is_symlink(const filesystem::path& 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 nocexcept;
  file_status status() const;
  file_status status(error_code& ec) const nocexcept;
  file_status status() const;
  file_status status(error_code& ec) const nocexcept;
  file_status symlink_status() const;
  file_status symlink_status(error_code& ec) const nocexcept;
};
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 available and storing the values would allow the implementation to eliminate file system accesses by directory_entry observer functions (30.11.14). Such stored file attribute values are said to be cached.

[ Note: For purposes of exposition, class directory_iterator (30.11.12) 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;

    // 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 ]

30.11.11.1 directory_entry constructors

    explicit directory_entry(const filesystem::path& p);
directory_entry(const filesystem::path& p, error_code& ec);

1 Effects: Constructs an object of type directory_entry, then refresh() or refresh(ec), respectively.
2 Postconditions: path() == p if no error occurs, otherwise path() == filesystem::path().
3 Throws: As specified in 30.11.6.

30.11.11.2 directory_entry modifiers

void assign(const filesystem::path& p);
void assign(const filesystem::path& p, error_code& ec);

1 Effects: Equivalent to pathobject = p, then refresh() or refresh(ec), respectively. If an error occurs, the values of any cached attributes are unspecified.
2 Throws: As specified in 30.11.6.
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 30.11.6.

void refresh();
void refresh(error_code& ec) noexcept;

Effects: Stores the current values of any cached attributes of the file p resolves to. If an error occurs, an error is reported (30.11.6) and the values of any cached attributes are unspecified.

Throws: As specified in 30.11.6.

[Note: Implementations of directory_iterator (30.11.12) 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]

30.11.11.3 directory_entry observers

Unqualified function names in the Returns: elements of the directory_entry observers described below refer to members of the std::filesystem namespace.

const filesystem::path& path() const noexcept;
operator const filesystem::path&() const noexcept;

Returns: pathobject.

bool exists() const;
bool exists(error_code& ec) const noexcept;

Returns: exists(this->status()) or exists(this->status(ec)), respectively.

Throws: As specified in 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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 30.11.6.

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.

 Thrones: As specified in 30.11.6.

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.

30.11.12 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 (30.11.11) 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 (30.11.13). —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&;

// 30.11.12.1, member functions
directory_iterator() noexcept;
eduard directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
directory_iterator(const path& p, error_code& ec) noexcept;
directory_iterator(const path& p, directory_options options,
    error_code& ec) noexcept;
directory_iterator(const directory_iterator& rhs);
directory_iterator(directory_iterator&& rhs) noexcept;
~directory_iterator();
directory_iterator& operator=(const directory_iterator& rhs);
directory_iterator& operator=(directory_iterator&& rhs) noexcept;

const directory_entry& operator*() const;
const directory_entry* operator->() const;
directory_iterator& operator++();
directory_iterator& increment(error_code& ec) noexcept;

// other members as required by 27.2.3, input iterators
};
}

directory_iterator satisfies the requirements of an input iterator (27.2.3).

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.

The end iterator is not dereferenceable.

Two end iterators are always equal. An end iterator shall not be equal to a non-end iterator.

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=/.

Directory iteration shall not yield directory entries for the current (dot) and parent (dot-dot) directories.

The order of directory entries obtained by dereferencing successive increments of a directory_iterator is unspecified.
Constructors and non-const `directory_iterator` member functions store the values of any cached attributes (30.11.11) 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 30.11.11 as a friend of class `directory_entry`. — end note]

[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]

[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]

### 30.11.12.1 `directory_iterator` members

```cpp
directory_iterator() noexcept;
```

**Effects:** Constructs the end iterator.

```cpp
explicit directory_iterator(const path& p);
directory_iterator(const path& p, directory_options options);
directory_iterator(const path& p, error_code& ec) noexcept;
directory_iterator(const path& p, directory_options options, error_code& ec) noexcept;
```

**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

```cpp
(options & directory_options::skip_permission_denied) != directory_options::none
```

and construction encounters an error indicating that permission to access `p` is denied, constructs the end iterator and does not report an error.

**Throws:** As specified in 30.11.6.

[Note: To iterate over the current directory, use `directory_iterator(".")` rather than `directory_iterator("""). — end note]

```cpp
directory_iterator(const directory_iterator& rhs);
directory_iterator(directory_iterator&& rhs) noexcept;
```

**Effects:** Constructs an object of class `directory_iterator`.

**Postconditions:** `*this` has the original value of `rhs`.

```cpp
directory_iterator& operator=(const directory_iterator& rhs);
directory_iterator& operator=(directory_iterator&& rhs) noexcept;
```

**Effects:** If `*this` and `rhs` are the same object, the member has no effect.

**Postconditions:** `*this` has the original value of `rhs`.

**Returns:** `*this`.

```cpp
directory_iterator& operator++();
directory_iterator& increment(error_code& ec) noexcept;
```

**Effects:** As specified for the prefix increment operation of Input iterators (27.2.3).

**Returns:** `*this`.

**Throws:** As specified in 30.11.6.

### 30.11.12.2 `directory_iterator` non-member functions

These functions enable range access for `directory_iterator`.

```cpp
directory_iterator begin(directory_iterator iter) noexcept;
```

**Returns:** `iter`.
directory_iterator end(const directory_iterator&) noexcept;

3

Returns: directory_iterator().

### 30.11.13 Class recursive_directory_iterator [fs.class.rec.dir.itr]

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.

```cpp
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&;

            // 30.11.13.1, constructors and destructor
            recursive_directory_iterator() noexcept;
            explicit recursive_directory_iterator(const path& p);
            recursive_directory_iterator(const path& p, directory_options options);
            recursive_directory_iterator(const path& p, directory_options options,
                error_code& ec) noexcept;
            recursive_directory_iterator(const path& p, error_code& ec) noexcept;
            recursive_directory_iterator(const recursive_directory_iterator& rhs);
            recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;
            ~recursive_directory_iterator();

            // 30.11.13.1, observers
            directory_options options() const;
            int depth() const;
            bool recursion_pending() const;
            const directory_entry& operator*() const;
            const directory_entry* operator->() const;

            // 30.11.13.1, 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) noexcept;

            void pop();
            void pop(error_code& ec);
            void disable_recursion_pending();

            // other members as required by 27.2.3, input iterators
    }
};
```

2 Calling options, depth, recursion_pending, pop or disable_recursion_pending on an iterator that is not dereferenceable results in undefined behavior.

3 The behavior of a `recursive_directory_iterator` is the same as a `directory_iterator` unless otherwise specified.

4 [Note: If the directory structure being iterated over contains cycles then the end iterator may be unreachable. —end note]
30.11.13.1 recursive_directory_iterator members

recursive_directory_iterator() noexcept;

Effects: Constructs the end iterator.

explicit recursive_directory_iterator(const path& p);
recursive_directory_iterator(const path& p, directory_options options);
recursive_directory_iterator(const path& p, directory_options options, error_code& ec) noexcept;
recursive_directory_iterator(const path& p, error_code& ec) noexcept;

Effects: Constructs a iterator representing the first entry in the directory p resolves to, if any; otherwise, the end iterator. However, if

(options & directory_options::skip_permission_denied) != directory_options::none

and construction encounters an error indicating that permission to access p is denied, constructs the end iterator and does not report an error.

Postconditions: options() == options for the signatures with a directory_options argument, otherwise options() == directory_options::none.

Throws: As specified in 30.11.6.

[Note: To iterate over the current directory, use recursive_directory_iterator(".") rather than recursive_directory_iterator("""). — end note]

recursive_directory_iterator(const recursive_directory_iterator& rhs);

Effects: Constructs an object of class recursive_directory_iterator.

Postconditions:
(8.1) options() == rhs.options()
(8.2) depth() == rhs.depth()
(8.3) recursion_pending() == rhs.recursion_pending()

recursive_directory_iterator(recursive_directory_iterator&& rhs) noexcept;

Effects: Constructs an object of class recursive_directory_iterator.

Postconditions: options(), depth(), and recursion_pending() have the values that rhs.options(), rhs.depth(), and rhs.recursion_pending(), respectively, had before the function call.

recursive_directory_iterator& operator=(const recursive_directory_iterator& rhs);

Effects: If *this and rhs are the same object, the member has no effect.

Postconditions:
(12.1) options() == rhs.options()
(12.2) depth() == rhs.depth()
(12.3) recursion_pending() == rhs.recursion_pending()

Returns: *this.

recursive_directory_iterator& operator=(recursive_directory_iterator&& rhs) noexcept;

Effects: If *this and rhs are the same object, the member has no effect.

Postconditions: options(), depth(), and recursion_pending() have the values that rhs.options(), rhs.depth(), and rhs.recursion_pending(), respectively, had before the function call.

Returns: *this.

directory_options options() const;

Returns: The value of the argument passed to the constructor for the options parameter, if present, otherwise directory_options::none.

Throws: Nothing.
int depth() const;

Returns: The current depth of the directory tree being traversed. [Note: 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) noexcept;

Effects: As specified for the prefix increment operation of Input iterators (27.2.3), except that:

(23.1) If there are no more entries at the current depth, then if depth() != 0 iteration over the parent directory resumes; otherwise *this = recursive_directory_iterator().

(23.2) Otherwise if recursion_pending() && is_directory((*this)->status()) && (!is_symlink((*this)->symlink_status()) || (options() & directory_options::follow_directory_symlink) != directory_options::none)
then either directory (*this)->path() is recursively iterated into or, if (options() & directory_options::skip_permission_denied) != directory_options::none
and an error occurs indicating that permission to access directory (*this)->path() is denied,
then directory (*this)->path() is treated as an empty directory and no error is reported.

Returns: *this.

Throws: As specified in 30.11.6.

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.

Throws: As specified in 30.11.6.

void disable_recursion_pending();

Postconditions: recursion_pending() == false.

[Note: disable_recursion_pending() is used to prevent unwanted recursion into a directory. —end note]

30.11.13.2 recursive_directory_iterator non-member functions [fs.rec.dir.itr.nonmembers]

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().

30.11.14 Filesystem operation functions [fs.op.funcs]

Filesystem operation functions query or modify files, including directories, in external storage.

[Note: Because hardware failures, network failures, file system races (30.11.2.3), 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 30.11.6. —end note]

30.11.14.1 Absolute [fs.op.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 (30.11.2.2).

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 30.11.6.

[ Note: To resolve symlinks, or perform other sanitization which might require queries to secondary storage, such as hard disks, consider canonical (30.11.14.2). — end note ]

[ Note: Implementations are strongly encouraged to not query secondary storage, and not consider !exists(p) an error. — 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 ]

30.11.14.2 Canonical
[ fs.op.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 30.11.6.

Remarks: !exists(p) is an error.

30.11.14.3 Copy
[ fs.op.copy ]

void copy(const path& from, const path& to);

Effects: Equivalent to copy(from, to, copy_options::none).

void copy(const path& from, const path& to, error_code& ec) noexcept;

Effects: Equivalent to copy(from, to, copy_options::none, ec).

void copy(const path& from, const path& to, copy_options options);
void copy(const path& from, const path& to, copy_options options, error_code& ec) noexcept;

Requires: At most one element from each option group (30.11.9.3) is set in options.

Effects: Before the first use of f and t:

(4.1) — If
    (options & copy_options::create_symlinks) != copy_options::none ||
    (options & copy_options::skip_symlinks) != copy_options::none
    then auto f = symlink_status(from) and if needed auto t = symlink_status(to).

(4.2) — Otherwise, if
    (options & copy_options::copy_symlinks) != copy_options::none
    then auto f = symlink_status(from) and if needed auto t = status(to).

(4.3) — Otherwise, auto f = status(from) and if needed auto t = status(to).

Effects are then as follows:

(4.4) — If f.type() or t.type() is an implementation-defined file type (30.11.9.2), then the effects are implementation-defined.

(4.5) — Otherwise, an error is reported as specified in 30.11.6 if:

(4.5.1) — exists(f) is false, or

(4.5.2) — equivalent(from, to) is true, or
is_other(f) || is_other(t) is true, or

is_directory(f) && is_regular_file(t) is true.

Otherwise, if is_symlink(f), then:

- If (options & copy_options::skip_symlinks) != copy_options::none then return.
- Otherwise if !exists(t) && (options & copy_options::copy_symlinks) != copy_options::none then copy_symlink(from, to).

Otherwise report an error as specified in 30.11.6.

Otherwise, if is_regular_file(f), then:

- If (options & copy_options::directories_only) != copy_options::none, then return.
- Otherwise, if (options & copy_options::create_symlinks) != copy_options::none, then create a symbolic link to the source file.
- Otherwise, if (options & copy_options::create_hard_links) != copy_options::none, then create a hard link to the source file.
- Otherwise, if is_directory(t), then copy_file(from, to/from.filename(), options).

Otherwise, copy_file(from, to, options).

Otherwise, if is_directory(f) && ((options & copy_options::recursive) != copy_options::none || options == copy_options::none) then:

- If exists(t) is false, then create_directory(to, from).
- Then, iterate over the files in from, as if by
  
  for (const directory_entry& x : directory_iterator(from))
  
  copy(x.path(), to/x.path().filename(), options | copy_options::unspecified)

- Otherwise, for the signature with argument ec, ec.clear().

- Otherwise, no effects.

Throws: As specified in 30.11.6.

Remarks: For the signature with argument ec, any library functions called by the implementation shall have an error_code argument if applicable.

Example: Given this directory structure:

```
/dir1
 file1
 file2
/dir2
 file3
```

Calling copy("/dir1", "/dir3") would result in:

```
/dir1
 file1
 file2
/dir2
 file3
```

Alternatively, calling copy("/dir1", "/dir3", copy_options::recursive) would result in:

```
/dir1
 file1
 file2
/dir2
```

5 Thros: As specified in 30.11.6.

6 Remarks: For the signature with argument ec, any library functions called by the implementation shall have an error_code argument if applicable.

7 Example: Given this directory structure:

```
/dir1
 file1
 file2
/dir2
 file3
```

Calling copy("/dir1", "/dir3") would result in:

```
/dir1
 file1
 file2
/dir2
 file3
```

Alternatively, calling copy("/dir1", "/dir3", copy_options::recursive) would result in:

```
/dir1
 file1
 file2
/dir2
```
30.11.14.4 Copy file

bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to, error_code& ec) noexcept;

1 Returns: copy_file(from, to, copy_options::none) or
copy_file(from, to, copy_options::none, ec), respectively.

2 Throws: As specified in 30.11.6.

bool copy_file(const path& from, const path& to, copy_options options);
bool copy_file(const path& from, const path& to, copy_options options,
error_code& ec) noexcept;

3 Requires: At most one element from each option group (30.11.9.3) is set in options.

4 Effects: As follows:

(4.1) — Report a file already exists error as specified in 30.11.6 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 (30.11.14.25).

(4.3) — Otherwise, no effects.

5 Returns: true if the from file was copied, otherwise false. The signature with argument ec returns false if an error occurs.

6 Throws: As specified in 30.11.6.

7 Complexity: At most one direct or indirect invocation of status(to).

30.11.14.5 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;

1 Effects: Equivalent to function(read_symlink(existing_symlink), new_symlink) or
function(read_symlink(existing_symlink, ec), new_symlink, ec), respectively, where in each
case function is create_symlink or create_directory_symlink as appropriate.

2 Throws: As specified in 30.11.6.

30.11.14.6 Create directories

bool create_directories(const path& p);
30.11.14.7 Create directory

bool create_directories(const path& p, error_code& ec) noexcept;

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. The signature with argument ec returns false if an error occurs.
Throws: As specified in 30.11.6.
Complexity: $O(n)$ where n is the number of elements of p.

30.11.14.8 Create directory symlink

void create_directory_symlink(const path& to, const path& new_symlink);
void create_directory_symlink(const path& to, const path& new_symlink, error_code& ec) noexcept;

Effects: Establishes the postcondition, as if by POSIX symlink().
Postconditions: new_symlink resolves to a symbolic link file that contains an unspecified representation of to.
Throws: As specified in 30.11.6.

[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]

30.11.14.9 Create hard link

void create_hard_link(const path& to, const path& new_hard_link);
void create_hard_link(const path& to, const path& new_hard_link, error_code& ec) noexcept;

Effects: Establishes the postcondition, as if by POSIX link().
Postconditions: exists(to) && exists(new_hard_link) && equivalent(to, new_hard_link)

§ 30.11.14.9 1146
The contents of the file or directory to resolves to are unchanged.

Throws: As specified in 30.11.6.

[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]

30.11.14.10 Create symlink

```cpp
void create_symlink(const path& to, const path& new_symlink);
void create_symlink(const path& to, const path& new_symlink,
              error_code& ec) noexcept;
```

Effects: Establishes the postcondition, as if by POSIX symlink().

Postconditions: new_symlink resolves to a symbolic link file that contains an unspecified representation of to.

Throws: As specified in 30.11.6.

[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]

30.11.14.11 Current path

```cpp
path current_path();
path current_path(error_code& ec);
```

Returns: The absolute path of the current working directory, whose pathname in the native format is obtained as if by POSIX getcwd(). The signature with argument ec returns path() if an error occurs.

Throws: As specified in 30.11.6.

Remarks: The current working directory is the directory, associated with the process, that is used as the starting location in pathname resolution for relative paths.

[Note: The current_path() name was chosen to emphasize that the returned value is a path, not just a single directory name. — end note]

[Note: The current path as returned by many operating systems is a dangerous global variable. It may be changed unexpectedly by a third-party or system library functions, or by another thread. — end note]

```cpp
void current_path(const path& p);
void current_path(const path& p, error_code& ec) noexcept;
```

Effects: Establishes the postcondition, as if by POSIX chdir().

Postconditions: equivalent(p, current_path()).

Throws: As specified in 30.11.6.

[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]

30.11.14.12 Equivalent

```cpp
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 structure, 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 30.11.6.
30.11.14.13  Exists

bool exists(file_status s) noexcept;

Returns: status_known(s) && s.type() != file_type::not_found.

bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;

Let s be a file_status, determined as if by status(p) or status(p, ec), respectively.

Effects: The signature with argument ec calls ec.clear() if status_known(s).

Returns: exists(s).
Throws: As specified in 30.11.6.

30.11.14  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 (30.11.6).

Returns:
(2.1) — If is_regular_file(p), the size in bytes of the file p resolves to, determined as if by the value of the POSIX stat structure member st_size obtained as if by POSIX stat().
(2.2) — Otherwise, the result is implementation-defined.

The signature with argument ec returns static_cast<uintmax_t>(-1) if an error occurs.

Throws: As specified in 30.11.6.

30.11.15  Hard link count

uintmax_t hard_link_count(const path& p);
uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;

Returns: The number of hard links for p. The signature with argument ec returns static_cast<uintmax_t>(-1) if an error occurs.

Throws: As specified in 30.11.6.

30.11.16  Is block file

bool is_block_file(file_status s) noexcept;

Returns: s.type() == file_type::block.

bool is_block_file(const path& p);
bool is_block_file(const path& p, error_code& ec) noexcept;

Returns: is_block_file(status(p)) or is_block_file(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 30.11.6.

30.11.17  Is character file

bool is_character_file(file_status s) noexcept;

Returns: s.type() == file_type::character.

bool is_character_file(const path& p);
bool is_character_file(const path& p, error_code& ec) noexcept;

Returns: is_character_file(status(p)) or is_character_file(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 30.11.6.
30.11.14.18  Is directory

bool is_directory(file_status s) noexcept;
1  
 Returns: s.type() == file_type::directory.

bool is_directory(const path& p);
bool is_directory(const path& p, error_code& ec) noexcept;
2  
 Returns: is_directory(status(p)) or is_directory(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 30.11.6.

30.11.14.19  Is empty

bool is_empty(const path& p);
bool is_empty(const path& p, error_code& ec) noexcept;
1  
 Effects:

(1.1) — Determine file_status s, as if by status(p) or status(p, ec), respectively.
(1.2) — For the signature with argument ec, return false if an error occurred.
(1.3) — Otherwise, if is_directory(s):
(1.3.1) — Create a variable itr, as if by directory_iterator itr(p) or directory_iterator itr(p, ec), respectively.
(1.3.2) — For the signature with argument ec, return false if an error occurred.
(1.3.3) — Otherwise, return itr == directory_iterator().
(1.4) — Otherwise:
(1.4.1) — Determine uintmax_t sz, as if by file_size(p) or file_size(p, ec), respectively.
(1.4.2) — For the signature with argument ec, return false if an error occurred.
(1.4.3) — Otherwise, return sz == 0.

2  
 Throws: As specified in 30.11.6.

30.11.14.20  Is fifo

bool is_fifo(file_status s) noexcept;
1  
 Returns: s.type() == file_type::fifo.

bool is_fifo(const path& p);
bool is_fifo(const path& p, error_code& ec) noexcept;
2  
 Returns: is_fifo(status(p)) or is_fifo(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 30.11.6.

30.11.14.21  Is other

bool is_other(file_status s) noexcept;
1  
 Returns: exists(s) && !is_regular_file(s) && !is_directory(s) && !is_symlink(s).

bool is_other(const path& p);
bool is_other(const path& p, error_code& ec) noexcept;
2  
 Returns: is_other(status(p)) or is_other(status(p, ec)), respectively. The signature with argument ec returns false if an error occurs.

Throws: As specified in 30.11.6.

30.11.14.22  Is regular file

bool is_regular_file(file_status s) noexcept;
1  
 Returns: s.type() == file_type::regular.
bool is_regular_file(const path& p);
2
Returns: is_regular_file(status(p)).
3
Throws: filesystem_error if status(p) would throw filesystem_error.

bool is_regular_file(const path& p, error_code& ec) noexcept;
4
Effects: Sets ec as if by status(p, ec). [Note: file_type::none, file_type::not_found and
file_type::unknown cases set ec to error values. To distinguish between cases, call the status
function directly. — end note]
5
Returns: is_regular_file(status(p, ec)). Returns false if an error occurs.

30.11.14.23 Is socket [fs.op.is_socket]
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 ec returns false if an error occurs.
3
Throws: As specified in 30.11.6.

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 ec returns false if an error occurs.
3
Throws: As specified in 30.11.6.

30.11.14.24 Is symlink [fs.op.is_symlink]
bool is_symlink(file_status s) noexcept;
1
Returns: s.type() == file_type::symlink.

bool is_symlink(const path& p);
bool is_symlink(const path& p, error_code& ec) noexcept;
2
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.
3
Throws: As specified in 30.11.6.

30.11.14.25 Last write time [fs.op.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;
1
Returns: The time of last data modification of p, determined as if by the value of the POSIX stat
structure member st_mtime obtained as if by POSIX stat(). The signature with argument ec returns
file_time_type::min() if an error occurs.
2
Throws: As specified in 30.11.6.

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;
3
Effects: Sets the time of last data modification of the file resolved to by p to new_time, as if by POSIX
futimens().
4
Throws: As specified in 30.11.6.
5
[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]

30.11.14.26 Permissions [fs.op.permissions]
void permissions(const path& p, perms prms, perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms, error_code& ec) noexcept;
void permissions(const path& p, perms prms, perm_options opts, error_code& ec);
1
Requires: Exactly one of the perm_options constants replace, add, or remove is present in opts.
2
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 chmodat().

[Note: Conceptually permissions are viewed as bits, but the actual implementation may use some other mechanism. — end note]

Throws: As specified in 30.11.6.

30.11.14.27 Proximate

path proximate(const path& p, error_code& ec);

1 Returns: proximate(p, current_path(), ec).

2 Throws: As specified in 30.11.6.

path proximate(const path& p, const path& base = current_path());

path proximate(const path& p, const path& base, error_code& ec);

3 Returns: For the first form:

weaklycanonical(p).lexically_proximate(weaklycanonical(base));

For the second form:

weaklycanonical(p, ec).lexically_proximate(weaklycanonical(base, ec));

or path() at the first error occurrence, if any.

4 Throws: As specified in 30.11.6.

30.11.14.28 Read symlink

path read_symlink(const path& p);

path read_symlink(const path& p, error_code& ec);

1 Returns: If p resolves to a symbolic link, a path object containing the contents of that symbolic link. The signature with argument ec returns path() if an error occurs.

2 Throws: As specified in 30.11.6. [Note: It is an error if p does not resolve to a symbolic link. — end note]

30.11.14.29 Relative

path relative(const path& p, error_code& ec);

1 Returns: relative(p, current_path(), ec).

2 Throws: As specified in 30.11.6.

path relative(const path& p, const path& base = current_path());

path relative(const path& p, const path& base, error_code& ec);

3 Returns: For the first form:

weaklycanonical(p).lexically_relative(weaklycanonical(base));

For the second form:

weaklycanonical(p, ec).lexically_relative(weaklycanonical(base, ec));

or path() at the first error occurrence, if any.

4 Throws: As specified in 30.11.6.

30.11.14.30 Remove

bool remove(const path& p);

bool remove(const path& p, error_code& ec) noexcept;

1 Effects: If exists(symlink_status(p, ec)), the file p is removed as if by POSIX remove(). [Note: A symbolic link is itself removed, rather than the file it resolves to. — end note]

2 Postconditions: exists(symlink_status(p)) is false.

3 Returns: false if p did not exist, otherwise true. The signature with argument ec returns false if an error occurs.
30.11.14.31 Remove all

uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec) noexcept;

Effects: Recursively deletes the contents of p if it exists, then deletes file p itself, as if by POSIX remove(). [Note: A symbolic link is itself removed, rather than the file it resolves to. — end note]

Postconditions: exists(symlink_status(p)) is false.

Returns: The number of files removed. The signature with argument ec returns static_cast<uintmax_t>(-1) if an error occurs.

Throws: As specified in 30.11.6.

30.11.14.32 Rename

void rename(const path& old_p, const path& new_p);
void rename(const path& old_p, const path& new_p, error_code& ec) noexcept;

Effects: Renames old_p to new_p, as if by POSIX rename().

[Note:

(1.1) — If old_p and new_p resolve to the same existing file, no action is taken.
(1.2) — Otherwise, the rename may include the following effects:

(1.2.1) — if new_p resolves to an existing non-directory file, new_p is removed; otherwise,
(1.2.2) — if new_p resolves to an existing directory, new_p is removed if empty on POSIX compliant operating systems but may be an error on other operating systems.

A symbolic link is itself renamed, rather than the file it resolves to. — end note]

Throws: As specified in 30.11.6.

30.11.14.33 Resize file

void resize_file(const path& p, uintmax_t new_size);
void resize_file(const path& p, uintmax_t new_size, error_code& ec) noexcept;

Postconditions: file_size(p) == new_size.

Throws: As specified in 30.11.6.

30.11.14.34 Space

space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;

Returns: An object of type space_info. The value of the space_info object is determined as if by using POSIX statvfs to obtain a POSIX struct statvfs, and then multiplying its f_blocks, f_bfree, and f_bavail members by its f_frsize member, and assigning the results to the capacity, free, and available members respectively. Any members for which the value cannot be determined shall be set to static_cast<uintmax_t>(-1). For the signature with argument ec, all members are set to static_cast<uintmax_t>(-1) if an error occurs.

Throws: As specified in 30.11.6.

Remarks: The value of member space_info::available is operating system dependent. [Note: available may be less than free. — end note]

30.11.14.35 Status

file_status status(const path& p);

Effects: As if:

error_code ec;
file_status result = status(p, ec);
if (result.type() == file_type::none)
    throw filesystem_error(implementation-supplied-message, p, ec);
return result;

Returns: See above.

Throws: filesystem_error. [Note: result values of file_status(file_type::not_found) and
file_status(file_type::unknown) are not considered failures and do not cause an exception to be
thrown. —end note]

file_status status(const path& p, error_code& ec) noexcept;

Effects: If possible, determines the attributes of the file p resolves to, as if by using POSIX stat()
to obtain a POSIX struct stat. If, during attribute determination, the underlying file system API
reports an error, sets ec to indicate the specific error reported. Otherwise, ec.clear(). [Note: This
allows users to inspect the specifics of underlying API errors even when the value returned by status() is
not file_status(file_type::none). —end note]

Let prms denote the result of (m & perms::mask), where m is determined as if by converting the
st_mode member of the obtained struct stat to the type perms.

Returns:

(6.1) If ec != error_code():

(6.1.1) If the specific error indicates that p cannot be resolved because some element of the path does
not exist, returns file_status(file_type::not_found).

(6.1.2) Otherwise, if the specific error indicates that p can be resolved but the attributes cannot be
determined, returns file_status(file_type::unknown).

(6.1.3) Otherwise, returns file_status(file_type::none).

[Note: These semantics distinguish between p being known not to exist, p existing but not being
able to determine its attributes, and there being an error that prevents even knowing if p exists.
These distinctions are important to some use cases. —end note]

(6.2) Otherwise,

(6.2.1) If the attributes indicate a regular file, as if by POSIX S_ISREG, returns file_status(file_-
type::regular, prms). [Note: file_type::regular implies appropriate <fstream> operations would succeed, assuming no hardware, permission, access, or file system race errors. Lack of file_type::regular does not necessarily imply <fstream> operations would fail on
a directory. —end note]

(6.2.2) Otherwise, if the attributes indicate a directory, as if by POSIX S_ISDIR, returns file_-status(file_type::directory, prms). [Note: file_type::directory implies that calling
directory_iterator(p) would succeed. —end note]

(6.2.3) Otherwise, if the attributes indicate a block special file, as if by POSIX S_ISBLK, returns
file_status(file_type::block, prms).

(6.2.4) Otherwise, if the attributes indicate a character special file, as if by POSIX S_ISCHR, returns
file_status(file_type::character, prms).

(6.2.5) Otherwise, if the attributes indicate a fifo or pipe file, as if by POSIX S_ISFIFO, returns
file_status(file_type::fifo, prms).

(6.2.6) Otherwise, if the attributes indicate a socket, as if by POSIX S_ISSOCK, returns file_-status(file_type::socket, prms).

(6.2.7) Otherwise, if the attributes indicate an implementation-defined file type (30.11.9.2), returns
file_status(file_type::A, prms), where A is the constant for the implementation-defined
file type.

(6.2.8) Otherwise, returns file_status(file_type::unknown, prms).

Remarks: If a symbolic link is encountered during pathname resolution, pathname resolution continues
using the contents of the symbolic link.
### 30.11.14.36 Status known

bool status_known(file_status s) noexcept;

**Returns:** \(s\text{.type()} \neq \text{file_type::none}\).

### 30.11.14.37 Symlink status

file_status symlink_status(const path& p);

**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 \(\text{prms}\) denote the result of \((m \& \text{perms::mask})\), 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, \text{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 30.11.6.

### 30.11.14.38 Temporary directory path

path temp_directory_path();

**Effects:** If `exists(p)` is `false` or `is_directory(p)` is `false`, an error is reported (30.11.6).

**Returns:** The path \(p\). The signature with argument `ec` returns `path()` if an error occurs.

**Throws:** As specified in 30.11.6.

**Example:** For POSIX-based operating systems, an implementation might return the path supplied by the first environment variable found in the list `TMPDIR`, `TMP`, `TEMP`, `TEMPDIR`, or if none of these are found, "/tmp".

For Windows-based operating systems, an implementation might return the path reported by the Windows `GetTempPath` API function. —end example]

### 30.11.14.39 Weakly canonical

path weakly_canonical(const path& p);

**Returns:** \(p\) with symlinks resolved and the result normalized (30.11.7.1).

**Effects:** Using `status(p)` or `status(p, ec)`, respectively, to determine existence, return a path composed by `operator/=` from the result of calling `canonical()` without a base argument and with a path argument composed of the leading elements of \(p\) that exist, if any, followed by the elements of \(p\) that do not exist, if any. For the first form, `canonical()` is called without an `error_code` argument. For the second form, `canonical()` is called with `ec` as an `error_code` argument, and `path()` is returned at the first error occurrence, if any.  

**Postconditions:** The returned path is in normal form (30.11.7.1).

**Remarks:** Implementations should avoid unnecessary normalization such as when `canonical` has already been called on the entirety of \(p\).

**Throws:** As specified in 30.11.6.

### 30.12 C library files

#### 30.12.1 Header `<cstdio>` synopsis

```cpp
namespace std {
    using size_t = see 21.2.4;
    using FILE = see below;
    using fpos_t = see below;
}
```


```c
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);
    FILE* freopen(const char* filename, const char* mode, FILE* stream);
    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 scanf(const char* format, ...);
    int sscanf(const char* s, const char* format, ...);
    int strncmp(const char* s1, const char* s2, size_t n);
    int strncpy(char* dest, const char* src, size_t n);
    int sscanf(const char* s, const char* format, va_list arg);
    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 putc(const char* s);
    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 fseek(FILE* stream, long int offset, int whence);
    int ftell(FILE* stream);
    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);
}
```

§ 30.12.1

© ISO/IEC

N4713
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 (20.5.5.9) with other calls to `tmpnam` with an argument that is a null pointer value.

See also: ISO C 7.21

### 30.12.2 Header `<cinttypes>` synopsis

```c
#include <cinttypes>  // see 21.4.1

namespace std {
  using imaxdiv_t = see below;
  intmax_t imaxabs(intmax_t j);
  imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
  intmax_t strtoimax(const char* nptr, char** endptr, int base);
  uintmax_t strtoumax(const char* nptr, char** endptr, int base);
  intmax_t wcstol(const wchar_t* nptr, wchar_t** endptr, int base);
  uintmax_t wcstoul(const wchar_t* nptr, wchar_t** endptr, int base);
  intmax_t abs(intmax_t);
  // optional, see below
  imaxdiv_t div(intmax_t, intmax_t);  // optional, see below
}
```

```c
#define PRIdN see below
#define PRIiN see below
#define PRIoN see below
#define PRIuN see below
#define PRIxN see below
#define PRIXN see below
#define SCNdN see below
#define SCNIN see below
#define SCNoN see below
#define SCNuN see below
#define SCNxN see below
#define PRIdLEASTN see below
#define PRIiLEASTN see below
#define PRIoLEASTN see below
#define PRIuLEASTN see below
#define PRIxLEASTN see below
#define PRIXLEASTN see below
#define SCNdLEASTN see below
#define SCNINLEASTN see below
#define SCNoLEASTN see below
#define SCNuLEASTN see below
#define SCNxLEASTN see below
#define PRIdFASTN see below
#define PRIiFASTN see below
#define PRIoFASTN see below
#define PRIuFASTN see below
#define PRIxFASTN see below
#define PRIXFASTN see below
#define SCNdFASTN see below
#define SCNINFASTN see below
#define SCNoFASTN see below
#define SCNuFASTN see below
#define SCNxFASTN see below
#define PRIdMAX see below
#define PRIiMAX see below
#define PRIoMAX see below
#define PRIuMAX see below
#define PRIxMAX see below
#define PRIXMAX see below
#define SCNdMAX 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 `<cstdint>` instead of `<stdint.h>`, and
2. 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
31 Regular expressions library [re]

31.1 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 (24.1) 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 described in Table 128.

Table 128 — Regular expressions library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31.2 Definitions</td>
<td></td>
</tr>
<tr>
<td>31.3 Requirements</td>
<td></td>
</tr>
<tr>
<td>31.5 Constants</td>
<td></td>
</tr>
<tr>
<td>31.6 Exception type</td>
<td></td>
</tr>
<tr>
<td>31.7 Traits</td>
<td></td>
</tr>
<tr>
<td>31.8 Regular expression template &lt;regex&gt;</td>
<td></td>
</tr>
<tr>
<td>31.9 Submatches</td>
<td></td>
</tr>
<tr>
<td>31.10 Match results</td>
<td></td>
</tr>
<tr>
<td>31.11 Algorithms</td>
<td></td>
</tr>
<tr>
<td>31.12 Iterators</td>
<td></td>
</tr>
<tr>
<td>31.13 Grammar</td>
<td></td>
</tr>
</tbody>
</table>

31.2 Definitions [re.def]
1 The following definitions shall apply to this Clause:

31.2.1 collating element [defns.regex.collating.element]
a sequence of one or more characters within the current locale that collate as if they were a single character.

31.2.2 finite state machine [defns.regex.finite.state.machine]
an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

31.2.3 format specifier [defns.regex.format.specifier]
a sequence of one or more characters that is to be replaced with some part of a regular expression match.

31.2.4 matched [defns.regex.matched]
a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

31.2.5 primary equivalence class [defns.regex.primary.equivalence.class]
a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accents, case, or locale specific tailorings.
31.2.6 [defns.regex.regular.expression]  
**regular expression**  
a pattern that selects specific strings from a set of character strings.

31.2.7 [defns.regex.subexpression]  
**sub-expression**  
a subset of a regular expression that has been marked by parenthesis.

31.3 Requirements [re.req]

1. This subclause defines requirements on classes representing regular expression traits. [Note: The class template \texttt{regex_traits}, defined in 31.7, satisfies these requirements. — end note]

2. The class template \texttt{basic_regex}, defined in 31.8, 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 \texttt{typedef-names} and functions in the template parameter \texttt{traits} used by the \texttt{basic_regex} class template. This subclause defines the semantics of these members.

3. To specialize class template \texttt{basic_regex} for a character container \texttt{CharT} and its related regular expression traits class \texttt{Traits}, use \texttt{basic_regex<CharT, Traits>}. 

4. In Table 129 \texttt{X} denotes a traits class defining types and functions for the character container type \texttt{charT}; \texttt{u} is an object of type \texttt{X}; \texttt{v} is an object of type \texttt{const X}; \texttt{p} is a value of type \texttt{const charT*}; \texttt{I1} and \texttt{I2} are input iterators (27.2.3); \texttt{F1} and \texttt{F2} are forward iterators (27.2.5); \texttt{c} is a value of type \texttt{const charT*}; \texttt{s} is an object of type \texttt{X::string_type}; \texttt{I} is a value of type \texttt{int}; \texttt{cl} is an object of type \texttt{X::char_class_type}, and \texttt{loc} is an object of type \texttt{X::locale_type}.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return type</th>
<th>Assertion/note pre-/post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{X::char_type}</td>
<td>\texttt{charT}</td>
<td>The character container type used in the implementation of class template \texttt{basic_regex}.</td>
</tr>
<tr>
<td>\texttt{X::string_type}</td>
<td>\texttt{basic_string&lt;charT&gt;}</td>
<td></td>
</tr>
<tr>
<td>\texttt{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>\texttt{X::char_class_type}</td>
<td>A bitmask type (20.4.2.1.4).</td>
<td>A bitmask type representing a particular character classification.</td>
</tr>
<tr>
<td>\texttt{X::length(p)}</td>
<td>\texttt{size_t}</td>
<td>Yields the smallest (i) such that (p[i] == 0). Complexity is linear in (i).</td>
</tr>
<tr>
<td>\texttt{v.translate(c)}</td>
<td>\texttt{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>\texttt{v.translate_nocase(c)}</td>
<td>\texttt{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>\texttt{v.transform(F1, F2)}</td>
<td>\texttt{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>
## 31.4 Header `<regex>` synopsis

```
#include <initializer_list>

namespace std {
    // 31.5, regex constants
    namespace regex_constants {
        using syntax_option_type = T1;
        using match_flag_type = T2;
        using error_type = T3;
    }

    // 31.6, class regex_error
    class regex_error;
}
```
/** 31.7. class template regex_traits **/ template<class charT> struct regex_traits;  

/** 31.8. class template basic_regex **/ template<class charT, class traits = regex_traits<charT>> class basic_regex; 

using regex = basic_regex<char>;  
using regex = basic_regex<wchar_t>; 

/** 31.8.6, basic_regex swap **/ template<class charT, class traits>  
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2); 

/** 31.9. class template sub_match **/ template<class BidirectionalIterator>  
    class sub_match; 

using csub_match = sub_match<const char*>; 
using wcsu_sub_match = sub_match< const wchar_t >; 
using assub_match = sub_match< string::const_iterator >; 
using wssub_match = sub_match< wstring::const_iterator >; 

/** 31.9.2, 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>  
    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); 

§ 31.4 1161
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);

§ 31.4 1162
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);

// 31.10, 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>;

// 31.10.7, match_results swap
template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
match_results<BidirectionalIterator, Allocator>& m2);
// 31.11.2. function template regex_match

```cpp
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);
```

```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);
```

```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);
```

```cpp
template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>&&,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,
                              Allocator>&, const basic_regex<charT, traits>&,
                 regex_constants::match_flag_type = regex_constants::match_default) = delete;
```

```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);
```

```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);
```

// 31.11.3. function template 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);
```

```cpp
template<class charT, class Allocator, class traits>
bool regex_search(const charT* str,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);
```

```cpp
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);
```

```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);
```

```cpp
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);
```

```cpp
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);
```

§ 31.4
template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>&&,
    match_results<typename basic_string<charT, ST, SA>::const_iterator,
    Allocator>&,
    const basic_regex<charT, traits>&, regex_constants::match_flag_type
    = regex_constants::match_default) = delete;

// 31.11.4, 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>
OutputIterator
regex_replace(OutputIterator out,
    BidirectionalIterator first, BidirectionalIterator last,
    const basic_regex<charT, traits>& e,
    const charT* fmt,
    regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT, class ST, class SA, class FST, class FSA>
basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    const basic_string<charT, FST, FSA>& fmt,
    regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT, class ST, class SA>
basic_string<charT, ST, SA>
regex_replace(const charT* s,
    const basic_regex<charT, traits>& e,
    const charT* fmt,
    regex_constants::match_flag_type flags = regex_constants::match_default);

§ 31.4 1165

// 31.12.1, class template regex_iterator
template<class BidirectionalIterator,
    class charT = typename iterator_traits<BidirectionalIterator>::value_type,
    class traits = regex_traits<charT>>
class regex_iterator;
using cregex_iterator = regex_iterator<const char*>;
using wcregex_iterator = regex_iterator<const wchar_t*>;
using sregex_iterator = regex_iterator<string::const_iterator>;
using wregex_iterator = regex_iterator<wstring::const_iterator>;

// 31.12.2, class template regex_token_iterator
template<class BidirectionalIterator,
    class charT = typename iterator_traits<BidirectionalIterator>::value_type,
    class traits = regex_traits<charT>>
31.5 Namespace `std::regex_constants` [re.const]

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.

31.5.1 Bitmask type `syntax_option_type` [re.synopt]

namespace std::regex_constants {
    using syntax_option_type = T1;
    inline constexpr syntax_option_type icase = unspecified;
    inline constexpr syntax_option_type nosubs = unspecified;
    inline constexpr syntax_option_type optimize = unspecified;
    inline constexpr syntax_option_type collate = unspecified;
    inline constexpr syntax_option_type ECMAScript = unspecified;
    inline constexpr syntax_option_type basic = unspecified;
    inline constexpr syntax_option_type extended = unspecified;
    inline constexpr syntax_option_type awk = unspecified;
    inline constexpr syntax_option_type grep = unspecified;
    inline constexpr syntax_option_type egrep = unspecified;
    inline constexpr syntax_option_type multiline = unspecified;
}

The type `syntax_option_type` is an implementation-defined bitmask type (20.4.2.1.4). Setting its elements has the effects listed in Table 130. 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.

31.5.2 Bitmask type `match_flag_type` [re.matchflag]

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;
}
Table 130 — 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 sequence shall be performed without regard to case.</td>
</tr>
<tr>
<td>nosubs</td>
<td>Specifies that no sub-expressions shall be considered to be marked, so that when a regular expression is matched against a character container sequence, no sub-expression matches shall be stored in the supplied match_results structure.</td>
</tr>
<tr>
<td>optimize</td>
<td>Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output.</td>
</tr>
<tr>
<td>collate</td>
<td>Specifies that character ranges of the form &quot;[a-b]&quot; shall be locale sensitive.</td>
</tr>
<tr>
<td>ECMAScript</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by ECMAScript in ECMA-262, as modified in 31.13.</td>
</tr>
<tr>
<td>basic</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by basic regular expressions in POSIX, Base Definitions and Headers, Section 9, Regular Expressions.</td>
</tr>
<tr>
<td>extended</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by extended regular expressions in POSIX, Base Definitions and Headers, Section 9, Regular Expressions.</td>
</tr>
<tr>
<td>awk</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by the utility awk in POSIX.</td>
</tr>
<tr>
<td>grep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep in POSIX.</td>
</tr>
<tr>
<td>egrep</td>
<td>Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep when given the -E option in POSIX.</td>
</tr>
<tr>
<td>multiline</td>
<td>Specifies that $ shall match the end of a line, if the ECMAScript engine is selected.</td>
</tr>
</tbody>
</table>

1 The type match_flag_type is an implementation-defined bitmask type (20.4.2.1.4). The constants of that type, except for match_default and format_default, are bitmask elements. The match_default and format_default constants are empty bitmasks. Matching a regular expression against a sequence of characters [first, last) proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in Table 131 for any bitmask elements set.

Table 131 — 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 it is not at the beginning of a line, so the character ^ in the regular 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 it is not at the end of a line, so the character &quot;$&quot; in the regular expression shall not match [last, last).</td>
</tr>
<tr>
<td>match_not_bow</td>
<td>The expression \b shall not match the sub-sequence [first, first).</td>
</tr>
<tr>
<td>match_not_eow</td>
<td>The expression \b shall not match the sub-sequence [last, last).</td>
</tr>
<tr>
<td>match_any</td>
<td>If more than one match is possible then any match is an acceptable result.</td>
</tr>
<tr>
<td>match_not_null</td>
<td>The expression shall not match an empty sequence.</td>
</tr>
<tr>
<td>match_continuous</td>
<td>The expression shall only match a sub-sequence that begins at first.</td>
</tr>
<tr>
<td>match_prev_avail</td>
<td>--first is a valid iterator position. When this flag is set the flags match_not_bol and match_not_bow shall be ignored by the regular expression algorithms (31.11) and iterators (31.12).</td>
</tr>
</tbody>
</table>
Table 131 — `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><code>format_default</code></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><code>format_sed</code></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 <code>sed</code> utility in POSIX.</td>
</tr>
<tr>
<td><code>format_no_copy</code></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><code>format_first_only</code></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>

31.5.3 Implementation-defined `error_type`[re.err]

```cpp
class regex_constants
{
    using error_type = T3;
    inline constexpr error_type error_collate = unspecified;
    inline constexpr error_type error_ctype = unspecified;
    inline constexpr error_type error_escape = unspecified;
    inline constexpr error_type error_backref = unspecified;
    inline constexpr error_type error_brack = unspecified;
    inline constexpr error_type error_paren = unspecified;
    inline constexpr error_type error_brace = unspecified;
    inline constexpr error_type error_badbrace = unspecified;
    inline constexpr error_type error_range = unspecified;
    inline constexpr error_type error_space = unspecified;
    inline constexpr error_type error_badrepeat = unspecified;
    inline constexpr error_type error_complexity = unspecified;
    inline constexpr error_type error_stack = unspecified;
}
```

The type `error_type` is an implementation-defined enumerated type (20.4.2.1.3). Values of type `error_type` represent the error conditions described in Table 132:

Table 132 — `error_type` values in the C locale

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>error_collate</code></td>
<td>The expression contained an invalid collating element name.</td>
</tr>
<tr>
<td><code>error_ctype</code></td>
<td>The expression contained an invalid character class name.</td>
</tr>
<tr>
<td><code>error_escape</code></td>
<td>The expression contained an invalid escaped character, or a trailing escape.</td>
</tr>
<tr>
<td><code>error_backref</code></td>
<td>The expression contained an invalid back reference.</td>
</tr>
<tr>
<td><code>error_brack</code></td>
<td>The expression contained mismatched <code>[</code> and <code>]</code>.</td>
</tr>
<tr>
<td><code>error_paren</code></td>
<td>The expression contained mismatched <code>{</code> and <code>}</code>.</td>
</tr>
<tr>
<td><code>error_brace</code></td>
<td>The expression contained mismatched <code>{</code> and <code>}</code>.</td>
</tr>
<tr>
<td><code>error_badbrack</code></td>
<td>The expression contained an invalid range in a <code>{</code> expression.</td>
</tr>
<tr>
<td><code>error_range</code></td>
<td>The expression contained an invalid character range, such as <code>[b-a]</code> in most encodings.</td>
</tr>
<tr>
<td><code>error_space</code></td>
<td>There was insufficient memory to convert the expression into a finite state machine.</td>
</tr>
<tr>
<td><code>error_badrepeat</code></td>
<td>One of *?++{ was not preceded by a valid regular expression.</td>
</tr>
<tr>
<td><code>error_complexity</code></td>
<td>The complexity of an attempted match against a regular expression exceeded a pre-set level.</td>
</tr>
</tbody>
</table>
### Table 132 — error_type values in the C locale (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Error condition</th>
</tr>
</thead>
<tbody>
<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>

#### 31.6 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`.

3. **Postconditions:** `ecode == code()`.

4. **Returns:** The error code that was passed to the constructor.

#### 31.7 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 (31.3).

```cpp
using char_class_type = bitmask_type;
```

2. The type `char_class_type` is used to represent a character classification and is capable of holding an implementation specific set returned by `lookup_classname`.

§ 31.7
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.begin(), &*str.begin() + 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.\footnote{332} 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 133 shall be recognized. For regex_traits<wchar_t>, at least the wide character names in Table 133 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:
   //   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 133 is set, then the result is determined as if by:
   //    ctype_base::mask m = convert<charT>(f);
   //    const ctype<charT>& ct = use_facet<ctype<charT>>(getloc());

\footnote{332} For example, if the parameter icase is true then [[:lower:]] is the same as [[:alpha:]].
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
t.isctype('_', f); // returns true
t.isctype(' ', f); // returns false
— end example]

int value(charT ch, int radix) const;

Requires: The value of radix shall be 8, 10, or 16.

Returns: The value represented by the digit ch in base radix if the character ch is a valid digit in base radix; otherwise returns -1.

locale_type imbue(locale_type loc);

Effects: Imbues this with a copy of the locale loc. [Note: Calling imbue with a different locale than the one currently in use invalidates all cached data held by *this. — end note]

Returns: If no locale has been previously imbued then a copy of the global locale in effect at the time of construction of *this, otherwise a copy of the last argument passed to imbue.

Postconditions: getloc() == loc.

locale_type getloc() const;

Returns: If no locale has been imbued then a copy of the global locale in effect at the time of construction of *this, otherwise a copy of the last argument passed to imbue.

31.8 Class template basic_regex

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 31.8, 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.

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]

The functions described in this Clause report errors by throwing exceptions of type regex_error.
Table 133 — Character class names and corresponding `ctype` masks

<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>

// 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;

// 31.8.1, constants
static constexpr regex_constants::syntax_option_type icase = regex_constants::icase;
static constexpr regex_constants::syntax_option_type nosubs = regex_constants::nosubs;
static constexpr regex_constants::syntax_option_type optimize = regex_constants::optimize;
static constexpr regex_constants::syntax_option_type collate = regex_constants::collate;
static constexpr regex_constants::syntax_option_type ECMAScript = regex_constants::ECMAScript;
static constexpr regex_constants::syntax_option_type basic = regex_constants::basic;
static constexpr regex_constants::syntax_option_type extended = regex_constants::extended;
static constexpr regex_constants::syntax_option_type awk = regex_constants::awk;
static constexpr regex_constants::syntax_option_type grep = regex_constants::grep;
static constexpr regex_constants::syntax_option_type egrep = regex_constants::egrep;
static constexpr regex_constants::syntax_option_type multiline = regex_constants::multiline;

// 31.8.2, construct/copy/destroy
basic_regex();
explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
basic_regex(const charT* p, size_t len, flag_type f = regex_constants::ECMAScript);
basic_regex(const basic_regex&);
basic_regex(basic_regex&&) noexcept;
template<class ST, class SA>
   explicit basic_regex(const basic_string<charT, ST, SA>& p,
                        flag_type f = regex_constants::ECMAScript);
template<class ForwardIterator>
    basic_regex(ForwardIterator first, ForwardIterator last,
        flag_type f = regex_constants::ECMAScript);

basic_regex(initializer_list<charT>, flag_type f = regex_constants::ECMAScript);
~basic_regex();

basic_regex& operator=(const basic_regex&);
basic_regex& operator=(basic_regex&&) noexcept;
basic_regex& operator=(const charT* ptr);

template<class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>& p);

// 31.8.3, 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);

template<class string_traits, class A>
    basic_regex& assign(const basic_string<charT, string_traits, A>& s,
        flag_type f = regex_constants::ECMAScript);

// 31.8.4, const operations
unsigned mark_count() const;
flag_type flags() const;

locale_type imbue(locale_type loc);
locale_type getloc() const;

// 31.8.6, swap
void swap(basic_regex&);
};

31.8.1 basic_regex constants

static constexpr regex_constants::syntax_option_type icase = regex_constants::icase;
static constexpr regex_constants::syntax_option_type nosubs = regex_constants::nosubs;
static constexpr regex_constants::syntax_option_type optimize = regex_constants::optimize;
static constexpr regex_constants::syntax_option_type collate = regex_constants::collate;
static constexpr regex_constants::syntax_option_type ECMAScript = regex_constants::ECMAScript;
static constexpr regex_constants::syntax_option_type basic = regex_constants::basic;
static constexpr regex_constants::syntax_option_type extended = regex_constants::extended;
static constexpr regex_constants::syntax_option_type awk = regex_constants::awk;
static constexpr regex_constants::syntax_option_type grep = regex_constants::grep;
static constexpr regex_constants::syntax_option_type egrep = regex_constants::egrep;
static constexpr regex_constants::syntax_option_type multiline = regex_constants::multiline;

1 The static constant members are provided as synonyms for the constants declared in namespace regex_constants.
31.8.2 basic_regex constructors

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.

Postconditions: 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);

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 the flags specified in f.

Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions within the expression.

basic_regex(const basic_regex& e);

Effects: Constructs an object of class basic_regex as a copy of the object e.

Postconditions: flags() and mark_count() return e.flags() and e.mark_count(), respectively.

basic_regex(basic_regex&& e) noexcept;

Effects: Move constructs an object of class basic_regex from e.

Postconditions: flags() and mark_count() return the values that e.flags() and e.mark_count(), respectively, had before construction. e is in a valid state with unspecified value.

template<class ST, class SA>
explicit basic_regex(const basic_string<charT, ST, SA>& s,
                      flag_type f = regex_constants::ECMAScript);

Throws: regex_error if s 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 string s, and interpreted according to the flags specified in f.

Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions within the expression.

template<class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
                      flag_type f = regex_constants::ECMAScript);

Throws: regex_error if the sequence [first, last) 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 [first, last), and interpreted according to the flags specified in f.

Postconditions: flags() returns f. mark_count() returns the number of marked sub-expressions within the expression.

basic_regex(initializer_list<charT> il, flag_type f = regex_constants::ECMAScript);

Effects: Same as basic_regex(il.begin(), il.end(), f).
31.8.3 basic_regex assign

basic_regex& operator=(const basic_regex& e);
   Effects: Copies e into *this and returns *this.
   Postconditions: flags() and mark_count() return e.flags() and e.mark_count(), respectively.

basic_regex& operator=(basic_regex&& e) noexcept;
   Effects: Move assigns from e into *this and returns *this.
   Postconditions: flags() and mark_count() return the values that e.flags() and e.mark_count(), respectively, had before assignment. e is in a valid state with unspecified value.

basic_regex& operator=(const charT* ptr);
   Requires: ptr shall not be a null pointer.
   Effects: Returns assign(ptr).

basic_regex& operator=(initializer_list<charT> il);
   Effects: Returns assign(il.begin(), il.end()).

   template<class ST, class SA>
      basic_regex& operator=(const basic_string<charT, ST, SA>& p);
         Effects: Returns assign(p).

   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.
   Postconditions: 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: The type InputIterator shall satisfy the requirements for an Input Iterator (27.2.3).
         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.
31.8.4 basic_regex 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.

31.8.5 basic_regex 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.

31.8.6 basic_regex swap

void swap(basic_regex& e);

Effects: Swaps the contents of the two regular expressions.

Postconditions: *this contains the regular expression that was in e, e contains the regular expression that was in *this.

Complexity: Constant time.

31.8.7 basic_regex non-member functions

31.8.7.1 basic_regex non-member swap

template<class charT, class traits>
void swap(basic_regex<charT, traits>& lhs, basic_regex<charT, traits>& rhs);

Effects: Calls lhs.swap(rhs).

31.9 Class template sub_match

Class template sub_match denotes the sequence of characters matched by a particular marked sub-expression.

namespace std {
    template<class BidirectionalIterator>
    class sub_match : public pair<BidirectionalIterator, BidirectionalIterator> {
    public:
        using value_type = typename iterator_traits<BidirectionalIterator>::value_type;
        using difference_type = typename iterator_traits<BidirectionalIterator>::difference_type;
        using iterator = BidirectionalIterator;
        using string_type = basic_string<value_type>;

        bool matched;

        constexpr sub_match();

        difference_type length() const;
        operator string_type() const;
        string_type str() const;

        int compare(const sub_match& s) const;
        int compare(const string_type& s) const;
        int compare(const value_type* s) const;
    }
31.9.1 sub_match members

constexpr sub_match();

Effects: Value-initializes the pair base class subobject and the member matched.

difference_type length() const;

Returns: matched ? distance(first, second) : 0.

operator string_type() const;

Returns: matched ? string_type(first, second) : string_type().

string_type str() const;

Returns: matched ? string_type(first, second) : string_type().

int compare(const sub_match& s) const;

Returns: str().compare(s.str()).

int compare(const string_type& s) const;

Returns: str().compare(s).

int compare(const value_type* s) const;

Returns: str().compare(s).

31.9.2 sub_match non-member operators

template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) == 0.

template<class BiIter>
bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) != 0.

template<class BiIter>
bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) < 0.

template<class BiIter>
bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) <= 0.

template<class BiIter>
bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);

Returns: lhs.compare(rhs) > 0.

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.
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);
8
Returns: !(lhs == 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);
9
Returns: 
    rhs.compare(typename sub_match<BiIter>::string_type(lhs.data(), lhs.size())) > 0

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);
10
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);
11
Returns: !(lhs < 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);
12
Returns: !(lhs == 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);
13
Returns: 
    lhs.compare(typename sub_match<BiIter>::string_type(rhs.data(), rhs.size())) == 0

template<class BiIter, class ST, class SA>
bool operator!=(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
14
Returns: !(lhs == 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);
15
Returns: 
    rhs.compare(typename sub_match<BiIter>::string_type(rhs.data(), rhs.size())) < 0

template<class BiIter, class ST, class SA>
bool operator>(const sub_match<BiIter>& lhs,
    const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
16
Returns: rhs < lhs.
const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

17 Returns: !(lhs < 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);

18 Returns: !(rhs < lhs).

template<class BiIter>
bool operator==(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

19 Returns: rhs.compare(lhs) == 0.

template<class BiIter>
bool operator!=(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

20 Returns: !(lhs == rhs).

template<class BiIter>
bool operator<(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

21 Returns: rhs.compare(lhs) > 0.

template<class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

22 Returns: rhs < lhs.

template<class BiIter>
bool operator<=(const typename iterator_traits<BiIter>::value_type* lhs,
    const sub_match<BiIter>& rhs);

23 Returns: !(lhs < rhs).

template<class BiIter>
bool operator==(const sub_match<BiIter>& lhs,
    const typename iterator_traits<BiIter>::value_type* rhs);

24 Returns: lhs.compare(rhs) == 0.

template<class BiIter>
bool operator!=(const sub_match<BiIter>& lhs,
    const typename iterator_traits<BiIter>::value_type* rhs);

25 Returns: !(lhs == rhs).

template<class BiIter>
bool operator<(const sub_match<BiIter>& lhs,
    const typename iterator_traits<BiIter>::value_type* rhs);

26 Returns: lhs.compare(rhs) < 0.

template<class BiIter>
bool operator>(const sub_match<BiIter>& lhs,
    const typename iterator_traits<BiIter>::value_type* rhs);

27 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: !(rhs < lhs).

template<class BiIter>
bool operator==(const typename iterator_traits<BiIter>::value_type& lhs,
const sub_match<BiIter>& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, lhs)) == 0.

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 typename iterator_traits<BiIter>::value_type& lhs,
const sub_match<BiIter>& rhs);

Returns: rhs.compare(typename sub_match<BiIter>::string_type(1, rhs)) > 0.

template<class BiIter>
bool operator>(const typename iterator_traits<BiIter>::value_type& lhs,
const sub_match<BiIter>& rhs);

Returns: rhs < lhs.

§ 31.9.2 1180
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 charT, class ST, class BiIter>
basic_ostream<charT, ST>&
operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

Returns: os << m.str().

### 31.10 Class template match_results

Class template `match_results` denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class template `match_results`.

The class template `match_results` satisfies the requirements of an allocator-aware container and of a sequence container (26.2.1, 26.2.3) except that only 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 {
        public:
            using value_type = sub_match<BidirectionalIterator>;
            using const_reference = const value_type&;
            using reference = value_type&;
            using const_iterator = implementation-defined;
            using iterator = const_iterator;
            using difference_type =
                typename iterator_traits<BidirectionalIterator>::difference_type;
            using size_type = typename allocator_traits<Allocator>::size_type;
            using allocator_type = Allocator;
            using char_type =
                typename iterator_traits<BidirectionalIterator>::value_type;
            using string_type = basic_string<char_type>;

            // 31.10.1, construct/copy/destroy
            explicit match_results(const Allocator& a = 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();
```
31.10.1 match_results constructors

1 In all match_results constructors, a copy of the Allocator argument shall be used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

match_results(const Allocator& a = Allocator());

Effects: Constructs an object of class match_results.

Postconditions: 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 134. Additionally, the stored Allocator value is move constructed from m.get_allocator().
6  **Throws:** Nothing.

match_results& operator=(const match_results& m);

7  **Effects:** Assigns m to *this. The postconditions of this function are indicated in Table 134.

match_results& operator=(match_results&& m);

8  **Effects:** Move-assigns m to *this. The postconditions of this function are indicated in Table 134.

Table 134 — 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>

31.10.2 match_results state

bool ready() const;

1  **Returns:** true if *this has a fully established result state, otherwise false.

31.10.3 match_results size

size_type size() const;

1  **Returns:** One plus the number of marked sub-expressions in the regular expression that was matched if *this represents the result of a successful match. Otherwise returns 0.  

[Note: The state of a match_results object can be modified only by passing that object to regex_match or regex_search. Sections 31.11.2 and 31.11.3 specify the effects of those algorithms on their match_results arguments.  

—end note]

size_type max_size() const;

2  **Returns:** The maximum number of sub_match elements that can be stored in *this.

[[nodiscard]] bool empty() const;

3  **Returns:** size() == 0.

31.10.4 match_results element access

difference_type length(size_type sub = 0) const;

1  **Requires:** ready() == true.

2  **Returns:** (*this)[sub].length().

difference_type position(size_type sub = 0) const;

3  **Requires:** ready() == true.

4  **Returns:** The distance from the start of the target sequence to (*this)[sub].first.

string_type str(size_type sub = 0) const;

5  **Requires:** ready() == true.

6  **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.

31.10.5 match_results 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 an Output Iterator (27.2.4).

Effects: Copies the character sequence [fmt_first, fmt_last) to OutputIter out. Replaces each format specifier or escape sequence in the copied range with either the character(s) it represents or the sequence of characters within *this to which it refers. The bitmasks specified in flags determine which format specifiers and escape sequences are recognized.

Returns: out.

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:

    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:

    format(back_inserter(result), fmt, flags);

Returns: result.

§ 31.10.5
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:
    format(back_inserter(result), fmt, fmt + char_traits<char_type>::length(fmt), flags);

Returns: result.

31.10.6 match_results 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.

31.10.7 match_results swap

void swap(match_results& that);

Effects: Swaps the contents of the two sequences.

Postconditions: *this contains the sequence of matched sub-expressions that were in that, that contains the sequence of matched sub-expressions that were in *this.

Complexity: Constant time.

template<class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

Effects: As if by m1.swap(m2).

31.10.8 match_results 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:

(1.1)  m1.empty() && m2.empty(), or
(1.2)  !m1.empty() && !m2.empty(), and the following conditions are satisfied:
(1.2.1)  m1.prefix() == m2.prefix(),
(1.2.2)  m1.size() == m2.size() && equal(m1.begin(), m1.end(), m2.begin()), and
(1.2.3)  m1.suffix() == m2.suffix().

[Note: The algorithm equal is defined in Clause 28. — end note]

template<class BidirectionalIterator, class Allocator>
bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);

Returns: !(m1 == m2).

31.11 Regular expression algorithms

31.11.1 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.
31.11.2 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 requirements of a Bidirectional Iterator (27.2.6).

**Effects:** Determines whether there is a match between the regular expression `e`, and all of the character sequence `[first, last)`. The parameter `flags` is used to control how the expression is matched against the character sequence. When determining if there is a match, only potential matches that match the entire character sequence are considered. Returns `true` if such a match exists, `false` otherwise. [Example:

```
std::regex re("Get|GetValue");
std::cmatch m;
regex_search("GetValue", m, re);           // returns true, and m[0] contains "Get"
regex_search("GetValues", m, re);         // returns true, and m[0] contains "GetValue"
regex_match("GetValues", m, re);          // returns true, and m[0] contains "Get"
regex_match("GetValues", m, re);          // returns false
```
— end example]

**Postconditions:** `m.ready() == true` in all cases. If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns 0 and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in Table 135.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td><code>1 + e.mark_count()</code></td>
</tr>
<tr>
<td>m.empty()</td>
<td><code>false</code></td>
</tr>
<tr>
<td>m.prefix().first</td>
<td><code>first</code></td>
</tr>
<tr>
<td>m.prefix().second</td>
<td><code>first</code></td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td><code>false</code></td>
</tr>
<tr>
<td>m.suffix().first</td>
<td><code>last</code></td>
</tr>
<tr>
<td>m.suffix().second</td>
<td><code>last</code></td>
</tr>
<tr>
<td>m.suffix().matched</td>
<td><code>false</code></td>
</tr>
<tr>
<td>m[0].first</td>
<td><code>first</code></td>
</tr>
<tr>
<td>m[0].second</td>
<td><code>last</code></td>
</tr>
<tr>
<td>m[0].matched</td>
<td><code>true</code></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 <code>n</code>. Alternatively, if sub-expression <code>n</code> did not participate in the match, then <code>last</code>.</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 <code>n</code>. Alternatively, if sub-expression <code>n</code> did not participate in the match, then <code>last</code>.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers <code>0 &lt; n &lt; m.size()</code>, <code>true</code> if sub-expression <code>n</code> participated in the match, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>

```
template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);
```

**Effects:** Behaves "as if" by constructing an instance of `match_results<BidirectionalIterator>` what, and then returning the result of `regex_match(first, last, what, e, flags).`
template<class charT, class Allocator, class traits>
bool regex_match(const charT* str,
    match_results<const charT*, Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_match(str, str + char_traits<charT>::length(str), m, e, flags).

template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
    match_results<typename basic_string<charT, ST, SA>::const_iterator,
        Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_match(s.begin(), s.end(), m, e, flags).

template<class charT, class traits>
bool regex_match(const charT* str,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_match(str, str + char_traits<charT>::length(str), e, flags)

template<class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Returns: regex_match(s.begin(), s.end(), e, flags).

31.11.3 regex_search

template<class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
    match_results<BidirectionalIterator, Allocator>& m,
    const basic_regex<charT, traits>& e,
    regex_constants::match_flag_type flags = regex_constants::match_default);

Requires: Type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator (27.2.6).

Effects: Determines whether there is some sub-sequence within [first, last) that matches the
regular expression e. The parameter flags is used to control how the expression is matched against
the character sequence. Returns true if such a sequence exists, false otherwise.

Postconditions: m.ready() == true in all cases. If the function returns false, then the effect on
parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise
the effects on parameter m are given in Table 136.

Table 136 — Effects of regex_search algorithm

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m.size()</td>
<td>1 + e.mark_count()</td>
</tr>
<tr>
<td>m.empty()</td>
<td>false</td>
</tr>
<tr>
<td>m.prefix().first</td>
<td>first</td>
</tr>
<tr>
<td>m.prefix().second</td>
<td>m[0].first</td>
</tr>
<tr>
<td>m.prefix().matched</td>
<td>m.prefix().first != m.prefix().second</td>
</tr>
<tr>
<td>m.suffix().first</td>
<td>m[0].second</td>
</tr>
<tr>
<td>m.suffix().second</td>
<td>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>
</tbody>
</table>
Table 136 — Effects of regex_search algorithm (continued)

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[n].first</td>
<td>For all integers $0 &lt; n &lt; m.size()$, the start of the sequence that matched sub-expression $n$. Alternatively, if sub-expression $n$ did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].second</td>
<td>For all integers $0 &lt; n &lt; m.size()$, the end of the sequence that matched sub-expression $n$. Alternatively, if sub-expression $n$ did not participate in the match, then last.</td>
</tr>
<tr>
<td>m[n].matched</td>
<td>For all integers $0 &lt; n &lt; m.size()$, true if sub-expression $n$ participated in the match, false otherwise.</td>
</tr>
</tbody>
</table>

template<class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

4 Returns: regex_search(str, str + char_traits<charT>::length(str), m, e, flags).

template<class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
match_results<typename basic_string<charT, ST, SA>::const_iterator,
Allocator>& m,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

5 Returns: regex_search(s.begin(), s.end(), m, e, flags).

Effects: Behaves “as if” by constructing an object what of type match_results<BidirectionalIterator> and returning regex_search(first, last, what, e, flags).

template<class charT, class traits>
bool regex_search(const charT* str,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

7 Returns: regex_search(str, str + char_traits<charT>::length(str), e, flags).

template<class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
regex_constants::match_flag_type flags = regex_constants::match_default);

8 Returns: regex_search(s.begin(), s.end(), e, flags).

31.11.4 regex_replace

template<class OutputIterator, class BidirectionalIterator,
class traits, class charT, class ST, class SA>
OutputIterator
regex_replace(OutputIterator out,
BidirectionalIterator first, BidirectionalIterator last,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);
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

\[
\text{regex_iterator<BidirectionalIterator, charT, traits> } i(\text{first, last, e, flags})
\]

and uses \(i\) to enumerate through all of the matches \(m\) of type match_results<BidirectionalIterator> that occur within the sequence \([\text{first, last})\). If no such matches are found and \(!\text{(flags & regex_constants::format_no_copy)}\), then calls

\[
\text{out} = \text{copy(\text{first, last, out})}
\]

If any matches are found then, for each such match:

(1.1) If \(!\text{(flags & regex_constants::format_no_copy)}\), calls

\[
\text{out} = \text{copy(\text{m.prefix().first, m.prefix().second, out})}
\]

(1.2) Then calls

\[
\text{out} = \text{m.format(out, fmt, flags)}
\]

for the first form of the function and

\[
\text{out} = \text{m.format(out, fmt, fmt + char_traits<charT>::length(fmt), flags)}
\]

for the second.

Finally, if such a match is found and \(!\text{(flags & regex_constants::format_no_copy)}\), calls

\[
\text{out} = \text{copy(last_m.suffix().first, last_m.suffix().second, out)}
\]

where \(\text{last_m}\) is a copy of the last match found. If \text{flags & regex_constants::format_first_only} is nonzero, then only the first match found is replaced.


2
Returns: out.

template<class traits, class charT, class ST, class SA, class FST, class FSA>

basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const basic_string<charT, FST, FSA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT, class ST, class SA>

basic_string<charT, ST, SA>
regex_replace(const basic_string<charT, ST, SA>& s,
const basic_regex<charT, traits>& e,
const charT* fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

Effects: Constructs an empty string result of type basic_string<charT, ST, SA> and calls:

\[
\text{regex_replace(back inserter(result), s.begin(), s.end(), e, fmt, flags)};
\]

Returns: result.

template<class traits, class charT, class ST, class SA>

basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const basic_string<charT, ST, SA>& fmt,
regex_constants::match_flag_type flags = regex_constants::match_default);

template<class traits, class charT>

basic_string<charT>
regex_replace(const charT* s,
const basic_regex<charT, traits>& e,
const charT* fmt,
Effects: Constructs an empty string result of type basic_string<
charT> and calls:

```
regex_replace(back_inserter(result), s, s + char_traits<charT>::length(s), e, fmt, flags);
```

Returns: result.

### 31.12 Regular expression iterators

#### 31.12.1 Class template regex_iterator

The class template `regex_iterator` is an iterator adaptor. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A `regex_iterator` uses `regex_search` to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time `operator++` is used, the iterator finds and stores a value of `match_results<BidirectionalIterator>`. If the end of the sequence is reached (`regex_search` returns `false`), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a const `match_results<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a const `match_results<BidirectionalIterator>*` is returned. It is impossible to store things into `regex_iterator`. 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 {
        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

31.12.1.1 regex_iterator constructors

regex_iterator();

Effects: Constructs an end-of-sequence iterator.

regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
const regex_type& re,
regex_constants::match_flag_type m = regex_constants::match_default);

Effects: Initializes begin and end to a and b, respectively, sets pregex to &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.

31.12.1.2 regex_iterator comparisons

bool operator==(const regex_iterator& right) const;

Returns: true if *this and right are both end-of-sequence iterators or if the following conditions all hold:

(1.1)  begin == right.begin,
(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).

31.12.1.3 regex_iterator indirection

const value_type& operator*() const;

Returns: match.

const value_type* operator->() const;

Returns: &match.

31.12.1.4 regex_iterator 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 \( \text{match}[i].\text{matched} \) is \text{true}, \text{match}.\text{position}(i) \) shall return \text{distance}(\text{begin}, \text{match}[i].\text{first})].

\[ \text{Note: This means that } \text{match}.\text{position}(i) \text{ gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to } \text{regex_search}. \quad \text{— end note} \]

It is unspecified how the implementation makes these adjustments.

\[ \text{Note: This means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of } \text{regex_search} \text{ will not be called. } \quad \text{— end note} \]

\text{regex_iterator} \text{ operator++(int);} \text{ Effects: As if by:}

\begin{verbatim}
regex_iterator \text{tmp} = \*\text{this};
++(\*\text{this});
\text{return} \text{tmp;}
\end{verbatim}

\text{31.12.2} \text{ Class template } \text{regex_token_iterator} \text{ [re.tokiter]}

\text{The class template } \text{regex_token_iterator} \text{ 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 } \text{sub_match} \text{ class template instance that represents what matched a particular sub-expression within the regular expression.}

\text{When class } \text{regex_token_iterator} \text{ 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.}

\text{After it is constructed, the iterator finds and stores a value } \text{regex_iterator}<\text{BidirectionalIterator}>\text{position and sets the internal count } \text{N} \text{ to zero. It also maintains a sequence } \text{subs} \text{ which contains a list of the sub-expressions which will be enumerated. Every time } \text{operator++} \text{ is used the count } \text{N} \text{ is incremented; if } \text{N} \text{ exceeds or equals } \text{subs}\text{.size()}, \text{ then the iterator increments member } \text{position} \text{ and sets count } \text{N} \text{ to zero.}

\text{If the end of sequence is reached (} \text{position} \text{ 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.}

\text{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 } \text{operator*} \text{ on an end-of-sequence iterator is not defined. For any other iterator value a } \text{const sub_match}<\text{BidirectionalIterator}>\text{&} \text{ is returned. The result of } \text{operator->} \text{ on an end-of-sequence iterator is not defined. For any other iterator value a } \text{const sub_match}<\text{BidirectionalIterator}>\text{&} \text{ is returned.}

\text{It is impossible to store things into } \text{regex_token_iterators}. \text{ Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.}

\begin{verbatim}
namespace std {
    template<class BidirectionalIterator, 
    class charT = typename iterator_traits<BidirectionalIterator>::value_type, 
    class traits = regex_traits<charT>>
    class regex_token_iterator {
    public:
        using regex_type = basic_regex<charT, traits>;
        using iterator_category = forward_iterator_tag;
        using value_type = sub_match<BidirectionalIterator>;
        using difference_type = ptrdiff_t;
        using pointer = const value_type*;
        using reference = const value_type&;

        regex_token_iterator();
    }
\end{verbatim}
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) = delete;

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,  
const regex_type& re,  
const vector<int>& submatches,  
regex_constants::match_flag_type m =  
regex_constants::match_default) = delete;

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,  
const regex_type& re,  
initializer_list<int> submatches,  
regex_constants::match_flag_type m =  
regex_constants::match_default) = delete;

template<size_t N>  
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,  
const regex_type& re,  
const int (&submatches)[N],  
regex_constants::match_flag_type m =  
regex_constants::match_default) = delete;

regex_token_iterator(const regex_token_iterator&);
regex_token_iterator& operator=(const regex_token_iterator&);

bool operator==(const regex_token_iterator& ) const;
bool operator!=(const regex_token_iterator& ) const;
const value_type* operator*() const;
const value_type* operator->() const;
regex_token_iterator& operator++();
regex_token_iterator operator++(int);

private:
using position_iterator =  
regex_iterator<BidirectionalIterator, charT, traits> ; // exposition only
position_iterator position;  // exposition only
const value_type* result;  // exposition only
value_type suffix;  // exposition only
size_t N;  // exposition only
vector<int> subs;  // exposition only

A suffix iterator is a regex_token_iterator object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member result holds a pointer to the data member suffix.

§ 31.12.2
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]`.

31.12.2.1 `regex_token_iterator` constructors

`regex_token_iterator();`

*Effects:* Constructs the end-of-sequence iterator.

`regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, int submatch = 0, regex_constants::match_flag_type m = regex_constants::match_default);`

`regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, const vector<int>& submatches, regex_constants::match_flag_type m = regex_constants::match_default);`

`regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, initializer_list<int> submatches, regex_constants::match_flag_type m = regex_constants::match_default);`

`template<size_t N> regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const regex_type& re, const int (&submatches)[N], regex_constants::match_flag_type m = regex_constants::match_default);`

*Requires:* Each of the initialization values of `submatches` shall be `>= -1`.

*Effects:* The first constructor initializes the member `subs` to hold the single value `submatch`. The second constructor initializes the member `subs` to hold a copy of the argument `submatches`. The third and fourth constructors initialize the member `subs` to hold a copy of the sequence of integer values pointed to by the iterator range `[submatches.begin(), submatches.end())` and `[&submatches, &submatches + N)`, respectively.

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.

31.12.2.2 `regex_token_iterator` comparisons

`bool operator==(const regex_token_iterator& right) const;`

*Returns:* `true` if `*this` and `right` are both end-of-sequence iterators, or if `*this` and `right` are both suffix iterators and `suffix == right.suffix`; otherwise returns `false` if `*this` or `right` is an end-of-sequence iterator or a suffix iterator. Otherwise returns `true` if `position == right.position`, `N == right.N`, and `subs == right.subs`. Otherwise returns `false`.

`bool operator!=(const regex_token_iterator& right) const;`

*Returns:* `!(*this == right)`.  

31.12.2.3 `regex_token_iterator` indirection

`const value_type& operator*() const;`

*Returns:* `*result`.  

§ 31.12.3 1194
const value_type* operator->() const;

Returns: result.

### 31.12.2.4 regex_token_iterator increment [re.tokiter.incr]

regex_token_iterator& operator++();

Effects: Constructs a local variable prev of type position_iterator, initialized with the value of position.

1. If *this is a suffix iterator, sets *this to an end-of-sequence iterator.
2. Otherwise, if N + 1 < subs.size(), increments N and sets result to the address of the current match.
3. 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.
4. 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).
5. Otherwise, sets *this to an end-of-sequence iterator.

Returns: *this

regex_token_iterator& operator++(int);

Effects: Constructs a copy tmp of *this, then calls ++(*(this).

Returns: tmp.

### 31.13 Modified ECMAScript regular expression grammar [re.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:

```plaintext
ClassAtom ::
    -
    ClassAtomNoDash
    ClassAtomExClass
    ClassAtomCollatingElement
    ClassAtomEquivalence

IdentityEscape ::
    SourceCharacter but not c
```

The following new productions are then added:

```plaintext
ClassAtomExClass ::
    [ ClassName ]

ClassAtomCollatingElement ::
    [. ClassName .]

ClassAtomEquivalence ::
    [= ClassName =]

ClassName ::
    ClassNameCharacter
    ClassNameCharacter ClassName

ClassNameCharacter ::
    SourceCharacter but not one of "." "=":" 
```

§ 31.13
The productions \texttt{ClassAtomExClass}, \texttt{ClassAtomCollatingElement} and \texttt{ClassAtomEquivalence} provide functionality equivalent to that of the same features in regular expressions in POSIX.

The regular expression grammar may be modified by any \texttt{regex\_constants::syntax\_option\_type} flags specified when constructing an object of type specialization of \texttt{basic\_regex} according to the rules in Table 130.

A \texttt{ClassName} production, when used in \texttt{ClassAtomExClass}, is not valid if \texttt{traits\_inst.lookup\_classname} returns zero for that name. The names recognized as valid \texttt{ClassName}s are determined by the type of the traits class, but at least the following names shall be recognized: \texttt{alnum}, \texttt{alpha}, \texttt{blank}, \texttt{cntrl}, \texttt{digit}, \texttt{graph}, \texttt{lower}, \texttt{print}, \texttt{punct}, \texttt{space}, \texttt{upper}, \texttt{xdigit}, \texttt{d}, \texttt{s}, \texttt{w}. In addition the following expressions shall be equivalent:

\begin{itemize}
  \item \texttt{\d} and \\
  \item \texttt{\D} and \\
  \item \texttt{\s} and \\
  \item \texttt{\S} and \\
  \item \texttt{\w} and \\
  \item \texttt{\W} and \\
\end{itemize}

A \texttt{ClassName} production when used in a \texttt{Class Atom Collating Element} production is not valid if the value returned by \texttt{traits\_inst.lookup\_collatename} for that name is an empty string.

The results from multiple calls to \texttt{traits\_inst.lookup\_classname} can be bitwise OR’ed together and subsequently passed to \texttt{traits\_inst.isctype}.

A \texttt{ClassName} production when used in a \texttt{ClassAtomEquivalence} production is not valid if the value returned by \texttt{traits\_inst.lookup\_collatename} for that name is an empty string or if the value returned by \texttt{traits\_inst.transform\_primary} for the result of the call to \texttt{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 \texttt{regex\_error}.

If the \texttt{CV} of a \texttt{UnicodeEscapeSequence} is greater than the largest value that can be held in an object of type \texttt{charT} the translator shall throw an exception object of type \texttt{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 \texttt{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 \texttt{match\_flag\_type} flags (31.5.2) specified when using the regular expression object in one of the regular expression algorithms (31.11). The behavior is also localized by interaction with the traits class template parameter as follows:

\begin{enumerate}
  \item During matching of a regular expression finite state machine against a sequence of characters, two characters \texttt{c} and \texttt{d} are compared using the following rules:
    \begin{enumerate}
      \item if (flags() & regex\_constants::icase) the two characters are equal if \texttt{traits\_inst.translate\_nocase(c)} == \texttt{traits\_inst.translate\_nocase(d)};
      \item otherwise, if flags() & regex\_constants::collate the two characters are equal if \texttt{traits\_inst.translate(c)} == \texttt{traits\_inst.translate(d)};
      \item otherwise, the two characters are equal if c == d.
    \end{enumerate}
  \item During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range \texttt{c1-c2} against a character \texttt{c} is conducted as follows: if \texttt{flags() & regex\_constants::collate} is false then the character \texttt{c} is matched if \texttt{c1 <= c && c <= c2}, otherwise \texttt{c} is matched in accordance with the following algorithm:
    \begin{verbatim}
    string_type str1 = string_type(1,
        flags() & icase ?
        traits\_inst.translate\_nocase(c1) : traits\_inst.translate(c1);
    \end{verbatim}
\end{enumerate}
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 \texttt{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 \texttt{c} is a member of a character class designated by an iterator range \texttt{[first, last)} if \texttt{traits\_inst.isctype(c, traits\_inst.lookup\_classname(first, last, flags() & icase))} is true.
32 Atomic operations library [atomics]

32.1 General [atomics.general]

1 This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

2 The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 137 — Atomics library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32.4</td>
<td></td>
</tr>
<tr>
<td>32.5</td>
<td></td>
</tr>
<tr>
<td>32.6</td>
<td></td>
</tr>
<tr>
<td>32.6.1</td>
<td></td>
</tr>
<tr>
<td>32.8</td>
<td></td>
</tr>
<tr>
<td>32.9</td>
<td></td>
</tr>
</tbody>
</table>

32.2 Header <atomic> synopsis [atomics.syn]

namespace std {

    // 32.4, order and consistency
    enum class memory_order : unspecified;
    template<class T>
        T kill_dependency(T y) noexcept;

    // 32.5, lock-free property
    #define ATOMIC_BOOL_LOCK_FREE unspecified
    #define ATOMIC_CHAR_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

    // 32.6, atomic
    template<class T> struct atomic;
    template<class T> struct atomic<T*>;

    // 32.7, non-member functions
    template<class T>
        bool atomic_is_lock_free(const volatile atomic<T>*) noexcept;
    template<class T>
        bool atomic_is_lock_free(const atomic<T>*) noexcept;
    template<class T>
        void atomic_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;

§ 32.2 1198
template<class T>
  void atomic_store_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
  void atomic_store_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
  T atomic_load(const volatile atomic<T>*) noexcept;
template<class T>
  T atomic_load(const atomic<T>*) noexcept;

template<class T>
  T atomic_load_explicit(const volatile atomic<T>*, memory_order) noexcept;
template<class T>
  T atomic_load_explicit(const atomic<T>*, memory_order) noexcept;

template<class T>
  T atomic_exchange(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_exchange(atomic<T>*, typename atomic<T>::value_type) noexcept;

template<class T>
  T atomic_exchange_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
template<class T>
  T atomic_exchange_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

template<class T>
  bool atomic_compare_exchange_weak(volatile atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type) noexcept;
template<class T>
  bool atomic_compare_exchange_weak(atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type) noexcept;

template<class T>
  bool atomic_compare_exchange_strong(volatile atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type) noexcept;
template<class T>
  bool atomic_compare_exchange_strong(atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type) noexcept;

template<class T>
  bool atomic_compare_exchange_weak_explicit(volatile atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type, memory_order, memory_order) noexcept;
template<class T>
  bool atomic_compare_exchange_weak_explicit(atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type, memory_order, memory_order) noexcept;

template<class T>
  bool atomic_compare_exchange_strong_explicit(volatile atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type, memory_order, memory_order) noexcept;
template<class T>
  bool atomic_compare_exchange_strong_explicit(atomic<T>*, typename atomic<T>::value_type*, typename atomic<T>::value_type, memory_order, memory_order) noexcept;

template<class T>
  T atomic_fetch_add(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_add(atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_add_explicit(volatile atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;
template<class T>
T atomic_fetch_add_explicit(atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;
template<class T>
T atomic_fetch_sub(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_sub(atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
T atomic_fetch_sub_explicit(volatile atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;
template<class T>
T atomic_fetch_sub_explicit(atomic<T>*, typename atomic<T>::difference_type, memory_order) noexcept;
template<class T>
T atomic_fetch_and(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_and(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_and_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
template<class T>
T atomic_fetch_and_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
template<class T>
T atomic_fetch_xor(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_xor(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
T atomic_fetch_xor_explicit(volatile atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;
template<class T>
T atomic_fetch_xor_explicit(atomic<T>*, typename atomic<T>::value_type, memory_order) noexcept;

// 32.6.1, initialization
#define ATOMIC_VAR_INIT(value) see below

// 32.3, type aliases
using atomic_bool = atomic<bool>;
using atomic_char = atomic<char>;
using atomic_schar = atomic<signed char>;
using atomic_uchar = atomic<unsigned char>;
using atomic_short = atomic<short>;
using atomic_ushort = atomic<unsigned short>;
using atomic_int = atomic<int>;
using atomic_uint = atomic<unsigned int>;
using atomic_long = atomic<long>;
using atomic_ulong = atomic<unsigned long>;
using atomic_llong = atomic<long long>;;
using atomic_u1long = atomic<unsigned long long>;
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>;

// 32.3. Type aliases [atomics.alias]

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.

32.4 Order and consistency [atomics.order]

namespace std {
enum class memory_order : unspecified {

§ 32.4
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`.

There shall be a single total order `S` on all `memory_order::seq_cst` operations, consistent with the “happens before” order and modification orders for all affected locations, such that each `memory_order::seq_cst` operation `B` that loads a value from an atomic object `M` observes one of the following values:

1. the result of the last modification `A` of `M` that precedes `B` in `S`, if it exists, or
2. if `A` exists, the result of some modification of `M` that is not `memory_order::seq_cst` and that does not happen before `A`, or
3. if `A` does not exist, the result of some modification of `M` that is not `memory_order::seq_cst`.

   [Note: Although it is not explicitly required that `S` include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the “happens before” ordering. — end note]

For an atomic operation `B` that reads the value of an atomic object `M`, if there is a `memory_order::seq_cst` fence `X` sequenced before `B`, then `B` observes either the last `memory_order::seq_cst` modification of `M` preceding `X` in the total order `S` or a later modification of `M` in its modification order.

For atomic operations `A` and `B` on an atomic object `M`, where `A` modifies `M` and `B` takes its value, if there is a `memory_order::seq_cst` fence `X` such that `A` is sequenced before `X` and `B` follows `X` in `S`, then `B` observes either the effects of `A` or a later modification of `M` in its modification order.

For atomic operations `A` and `B` on an atomic object `M`, where `A` modifies `M` and `B` takes its value, if there are `memory_order::seq_cst` fences `X` and `Y` such that `A` is sequenced before `X`, `Y` is sequenced before `B`, and `X` precedes `Y` in `S`, then `B` observes either the effects of `A` or a later modification of `M` in its modification order.

For atomic modifications `A` and `B` of an atomic object `M`, `B` occurs later than `A` in the modification order of `M` if:

1. there is a `memory_order::seq_cst` fence `X` such that `A` is sequenced before `X`, and `X` precedes `B` in `S`, or
2. there is a `memory_order::seq_cst` fence `Y` such that `Y` is sequenced before `B`, and `A` precedes `Y` in `S`, or
there are \texttt{memory\_order::seq\_cst} fences $X$ and $Y$ such that $A$ is sequenced before $X$, $Y$ is sequenced before $B$, and $X$ precedes $Y$ in $S$.

[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} operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, \texttt{memory\_order::seq\_cst} fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. — 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,

```
// Thread 1:
r1 = y.load(memory_order::relaxed);
x.store(r1, memory_order::relaxed);
// Thread 2:
r2 = x.load(memory_order::relaxed);
y.store(r2, memory_order::relaxed);
```

should not produce $r1 == r2 == 42$, since the store of 42 to $y$ is only possible if the store to $x$ stores 42, which circularly depends on the store to $y$ storing 42. Note that without this restriction, such an execution is possible. — end note]

[Note: The recommendation similarly disallows $r1 == r2 == 42$ in the following example, with $x$ and $y$ again initially zero:

```
// Thread 1:
r1 = x.load(memory_order::relaxed);
if (r1 == 42) y.store(42, memory_order::relaxed);
// Thread 2:
r2 = y.load(memory_order::relaxed);
if (r2 == 42) x.store(42, memory_order::relaxed);
```

— end note]

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.

```
template<class T>
T kill_dependency(T y) noexcept;
```

Effects: The argument does not carry a dependency to the return value (6.8.2).

Returns: $y$.

### 32.5 Lock-free property

The `ATOMIC_..._LOCK_FREE` macros indicate the lock-free property of the corresponding atomic types, with the signed and unsigned variants grouped together. The properties also apply to the corresponding (partial) specializations of the `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.

The function `atomic_is_lock_free` (32.6.1) 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.

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]

32.6 Class template atomic

```cpp
namespace std {
    template<class T> struct atomic {
        using value_type = T;
        static constexpr bool is_always_lock_free = implementation-defined;
        bool is_lock_free() const volatile noexcept;
        bool is_lock_free() const noexcept;
        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;
        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 be trivially copyable (6.7). [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 as its corresponding argument type. Specializations should have the same size whenever possible, as this reduces the effort required to port existing code. — end note]

32.6.1 Operations on atomic types

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;

```cpp
constexpr atomic(T desired) noexcept;
```

2 Effects: Leaves the atomic object in an uninitialized state. [Note: These semantics ensure compatibility with C. — end note]

```cpp
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]

Returns:

```
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)`.

```
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:

```
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) noexce$$
```

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:

```
bool compare_exchange_weak(T& expected, T desired, 
memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired, 
memory_order success, memory_order failure) noexcept;
bool compare_exchange_strong(T& expected, T desired, 
memory_order success, memory_order failure) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired, 
memory_order success, memory_order failure) noexce$$
```

Returns: Atomically returns the value pointed to by this immediately before the effects.
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 contents of the memory pointed to by this for equality with that previously retrieved from expected, and if true, replaces the contents of the memory 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 contents of the memory in expected are replaced by the value read from the memory 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 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. E.g. 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: The memcpy and memcmp semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with operator== if the underlying type has padding bits, trap bits, or alternate representations of the same value. —end note]
32.6.2 Specializations for integers

There are specializations of the atomic template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `char16_t`, `char32_t`, `wchar_t`, and any other types needed by the typedefs in the header `<cstdint>`. For each such integral type `integral`, the specialization `atomic<internal>` provides additional atomic operations appropriate to integral types. [Note: For the specialization `atomic<bool>`, see 32.6. —end note]

namespace std {
    template<> struct atomic<internal> {
        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) noexcect;
        integral load(memory_order = memory_order::seq_cst) const volatile noexcect;
        integral load(memory_order = memory_order::seq_cst) const noexcect;
        operator integral() const volatile noexcect;
        operator integral() const noexcect;
        integral exchange(integral, memory_order = memory_order::seq_cst) volatile noexcect;
        integral exchange(integral, memory_order = memory_order::seq_cst) noexcect;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order, memory_order) volatile noexcect;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order, memory_order) noexcect;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order, memory_order) volatile noexcect;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order, memory_order) noexcect;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order = memory_order::seq_cst) volatile noexcect;
        bool compare_exchange_weak(integral&, integral,
                                     memory_order = memory_order::seq_cst) noexcect;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order = memory_order::seq_cst) volatile noexcect;
        bool compare_exchange_strong(integral&, integral,
                                       memory_order = memory_order::seq_cst) noexcect;
        integral fetch_add(integral, memory_order = memory_order::seq_cst) volatile noexcect;
        integral fetch_add(integral, memory_order = memory_order::seq_cst) noexcect;
        integral fetch_sub(integral, memory_order = memory_order::seq_cst) volatile noexcect;
        integral fetch_sub(integral, memory_order = memory_order::seq_cst) noexcect;
        integral fetch_and(integral, memory_order = memory_order::seq_cst) volatile noexcect;
        integral fetch_and(integral, memory_order = memory_order::seq_cst) noexcect;
        integral fetch_or(integral, memory_order = memory_order::seq_cst) volatile noexcect;
        integral fetch_or(integral, memory_order = memory_order::seq_cst) noexcect;
        integral fetch_xor(integral, memory_order = memory_order::seq_cst) volatile noexcect;
        integral fetch_xor(integral, memory_order = memory_order::seq_cst) noexcect;
    };

    atomic() noexcect = default;
    atomic<atomic> noexcect;
    atomic(const atomic&) = delete;
    atomic(const atomic&) volatile = delete;
    integral operator=(integral) volatile noexcect;
    integral operator=(integral) noexcect;

    integral operator++(int) volatile noexcect;
    integral operator++(int) noexcect;
    integral operator--(int) volatile noexcect;
    integral operator--(int) noexcect;
    integral operator++() volatile noexcect;
    integral operator++() noexcect;
    integral operator--() volatile noexcect;

§ 32.6.2
The atomic integral 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 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) volatile noexcept;
T fetch_key(T operand, memory_order order = memory_order::seq_cst) noexcept;

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).

Returns: Atomically, the value pointed to by this immediately before the effects.

Remarks: For signed integer types, arithmetic is defined to use two’s complement representation. There are no undefined results.

T operator op=(T operand) volatile noexcept;
T operator op=(T operand) noexcept;

Effects: Equivalent to: return fetch_key(operand) op operand;

32.6.3 Specializations for floating-point types

There are specializations of the atomic template for the floating-point types float, double, and long double. For each such floating-point type floating-point, the specialization atomic<floating-point> provides additional atomic operations appropriate to floating-point types.

namespace std {
    template<> struct atomic<floating-point> {
        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(float*, float*, memory_order, memory_order) volatile noexcept;
bool compare_exchange_weak(float*, float*, memory_order, memory_order) noexcept;
bool compare_exchange_strong(float*, float*, memory_order, memory_order) volatile noexcept;
bool compare_exchange_strong(float*, float*, memory_order, memory_order) noexcept;
bool compare_exchange_weak(float*, float*, memory_order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(float*, float*, memory_order = memory_order_seq_cst) noexcept;
bool compare_exchange_strong(float*, float*, memory_order = memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(float*, float*, memory_order = memory_order_seq_cst) noexcept;

float fetch_add(float, memory_order = memory_order_seq_cst) volatile noexcept;
float fetch_add(float, memory_order = memory_order_seq_cst) noexcept;
float fetch_sub(float, memory_order = memory_order_seq_cst) volatile noexcept;
float fetch_sub(float, memory_order = memory_order_seq_cst) noexcept;

atomic() noexcept = default;
constexpr atomic(float) noexcept;
atomic(const atomic&) = delete;
atomic& operator=(const atomic&) = delete;
atomic& operator=(const atomic&) volatile = delete;

§ 32.6.3 1209
T operator op=(T operand) noexcept;

Effects: Equivalent to: return fetch_key(operand) op operand;

Remarks: If the result is not a representable value for its type (8.1) 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 (21.3.2). The floating-point environment (29.4) for atomic arithmetic operations on floating-point may be different than the calling thread’s floating-point environment.

32.6.4 Partial specialization for pointers

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 noexcept;
    bool is_lock_free() const volatile 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, T*, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T*& T, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T*& T, T*, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T*& T, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T*& T, T*, memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_weak(T*& T, T*, memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_strong(T*& T, T*, memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_strong(T*& 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;

§ 32.6.4
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>

4 `T* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) volatile noexcept;`

5 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]

6 Effects: Atomically replaces the value pointed to by this with the result of the computation applied to the value pointed to by this and the given operand. Memory is affected according to the value of order. These operations are atomic read-modify-write operations (6.8.2).

7 Returns: Atomically, the value pointed to by this immediately before the effects.

8 Remarks: The result may be an undefined address, but the operations otherwise have no undefined behavior.

```
T* operator op=(ptrdiff_t operand) volatile noexcept;
T* operator op=(ptrdiff_t operand) noexcept;
```

8 Effects: Equivalent to: return fetch_key(operand) op operand;

32.6.5 Member operators common to integers and pointers to objects

32.7 Non-member functions

1 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;
```
template<class T>
void atomic_init(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]

32.8 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(memory_order order = memory_order::seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) noexcept;
bool atomic_flag::test_and_set_explicit(volatile atomic_flag*, memory_order order) noexcept;
bool atomic_flag::test_and_set_explicit(atomic_flag*, memory_order 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 order) noexcept;
void atomic_flag::clear_explicit(atomic_flag*, memory_order 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.

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_explicit(volatile atomic_flag*, memory_order order) noexcept;
bool atomic_flag::test_and_set_explicit(atomic_flag*, memory_order order) 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).

6 Returns: Atomically, the value of the object immediately before the effects.
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;

7 Requires: The order argument shall not be memory_order::consume, memory_order::acquire, nor memory_order::acq_rel.

8 Effects: Atomically sets the value pointed to by object or by this to false. Memory is affected according to the value of order.

32.9 Fences [atomics.fences]

1 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 $A$.

extern "C" void atomic_thread_fence(memory_order order) noexcept;

5 Effects: Depending on the value of order, this operation:

(5.1) has no effects, if order == memory_order::relaxed;
(5.2) is an acquire fence, if order == memory_order::acquire or order == memory_order::consume;
(5.3) is a release fence, if order == memory_order::release;
(5.4) is both an acquire fence and a release fence, if order == memory_order::acq_rel;
(5.5) is a sequentially consistent acquire and release fence, if order == memory_order::seq_cst.

extern "C" void atomic_signal_fence(memory_order order) noexcept;

6 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.

7 [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]
33 Thread support library

33.1 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 140.

Table 140 — Thread support library summary

<table>
<thead>
<tr>
<th>Subclause</th>
<th>Header(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.2 Requirements</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>33.3 Threads</td>
<td>&lt;thread&gt;</td>
</tr>
<tr>
<td>33.4 Mutual exclusion</td>
<td>&lt;mutex&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;shared_mutex&gt;</td>
</tr>
<tr>
<td>33.5 Condition variables</td>
<td>&lt;condition_variable&gt;</td>
</tr>
<tr>
<td>33.6 Futures</td>
<td>&lt;future&gt;</td>
</tr>
</tbody>
</table>

33.2 Requirements

33.2.1 Template parameter names

Throughout this Clause, the names of template parameters are used to express type requirements. If a template parameter is named `Predicate`, `operator()` applied to the template argument shall return a value that is convertible to `bool`.

33.2.2 Exceptions

Some functions described in this Clause are specified to throw exceptions of type `system_error` (22.5.7). 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 20.5.5.12.

[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_condition` of `operation_not_permitted` (22.5) 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.

33.2.3 Native handles

Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation-defined. [Note: 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]

33.2.4 Timing specifications

Several functions described in this Clause take an argument to specify a timeout. These timeouts are specified as either a `duration` or a `time_point` type as specified in 23.17.

Implementations necessarily have some delay in returning from a timeout. Any overhead in interrupt response, function return, and scheduling induces a “quality of implementation” delay, expressed as duration $D_i$. Ideally, this delay would be zero. Further, any contention for processor and memory resources induces a “quality of management” delay, expressed as duration $D_m$. The delay durations may vary from timeout to timeout, but in all cases shorter is better.
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.\(^3\) 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. [Note: This specification may result in the total duration of the wait decreasing when measured against a steady clock. — end note]

\((4.2)\) — if \(C_a \leq C_t\), the waiting function should not time out until \(\text{Clock}::\text{now()}\) returns a time \(C_n \geq C_t\), i.e., waking at \(C_t + D_i + D_m\). [Note: When the clock is adjusted backwards, this specification 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. — end note]

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. [Note: Implementations should decrease the duration of the wait when the clock is adjusted forwards. — end note]

[Note: If the clock is not synchronized with a steady clock, e.g., a CPU time clock, these timeouts might not provide useful functionality. — end note]

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.

Implementation-provided clocks that are used for these functions shall meet the TrivialClock requirements (23.17.3).

A function that takes an argument which specifies a timeout will throw if, during its execution, a clock, time point, or time duration throws an exception. Such exceptions are referred to as timeout-related exceptions. [Note: Instantiations of clock, time point and duration types supplied by the implementation as specified in 23.17.7 do not throw exceptions. — end note]

### 33.2.5 Requirements for Lockable types

#### 33.2.5.1 In general

An execution agent is an entity such as a thread that may perform work in parallel with other execution agents. [Note: Implementations or users may introduce other kinds of agents such as processes or thread-pool tasks. — end note] The calling agent is determined by context, e.g., the calling thread that contains the call, and so on.

[Note: Some lockable objects are “agent oblivious” in that they work for any execution agent model because they do not determine or store the agent’s ID (e.g., an ordinary spin lock). — end note]

The standard library templates unique_lock (33.4.4.3), shared_lock (33.4.4.4), scoped_lock (33.4.4.2), lock_guard (33.4.4.1), lock, try_lock (33.4.5), and condition_variable_any (33.5.4) all operate on user-supplied lockable objects. The BasicLockable requirements, the Lockable requirements, and the TimedLockable requirements list the requirements imposed by these library types in order to acquire or release ownership of a lock by a given execution agent. [Note: The nature of any lock ownership and any synchronization it may entail are not part of these requirements. — end note]

#### 33.2.5.2 BasicLockable requirements

A type \(L\) meets the BasicLockable requirements if the following expressions are well-formed and have the specified semantics (\(m\) denotes a value of type \(L\)).

---

\(^3\) All implementations for which standard time units are meaningful must necessarily have a steady clock within their hardware implementation.
33.2.5.3 Lockable requirements

A type $L$ meets the Lockable requirements if it meets the BasicLockable requirements and the following expressions are well-formed and have the specified semantics ($m$ denotes a value of type $L$).

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.

33.2.5.4 TimedLockable requirements

A type $L$ meets the TimedLockable requirements if it meets the Lockable 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 (23.17.5), and $abs_{time}$ denotes a value of an instantiation of time_point (23.17.6)).

m.try_lock_for($rel_{time}$)

**Effects:** Attempts to acquire a lock for the current execution agent within the relative timeout (33.2.4) 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 (33.2.4) 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.

33.2.6 decay_copy

In several places in this Clause the operation $DECAY_{COPY}(x)$ is used. All such uses mean call the function $decay_{copy}(x)$ and use the result, where $decay_{copy}$ is defined as follows:

```cpp
template<class T> decay_t<T> decay_copy(T&& v)
{ return std::forward<T>(v); }
```

33.3 Threads

33.3 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*]

33.3.1 Header <thread> synopsis

```cpp
namespace std {
    class thread {
```
void swap(thread& x, thread& y) noexcept;

namespace this_thread {
    thread::id get_id() noexcept;
    void yield() noexcept;
    template<class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template<class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
}

33.3.2 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 33.2.3

        // 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& x) noexcept;
        bool joinable() const noexcept;
        void join();
        void detach();
        id get_id() const noexcept;
        native_handle_type native_handle();    // see 33.2.3

        // static members
        static unsigned int hardware_concurrency() noexcept;
    }
};

33.3.2.1 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;
}
bool operator>(thread::id x, thread::id y) noexcept;
bool operator>=(thread::id x, thread::id y) noexcept;

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& out, thread::id id);

// hash support
template<class T> struct hash;
template<> struct hash<thread::id>;

1 An object of type thread::id provides a unique identifier for each thread of execution and a single distinct value for all thread objects that do not represent a thread of execution (33.3.2). 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.

2 thread::id shall be a trivially copyable class (Clause 12). The library may reuse the value of a thread::id of a terminated thread that can no longer be joined.

3 [ Note: Relational operators allow thread::id objects to be used as keys in associative containers. — end note ]

id() noexcept;

4 Effects: Constructs an object of type id.

5 Postconditions: The constructed object does not represent a thread of execution.

bool operator==(thread::id x, thread::id y) noexcept;

6 Returns: true only if x and y represent the same thread of execution or neither x nor y represents a thread of execution.

bool operator!=(thread::id x, thread::id y) noexcept;

7 Returns: !(x == y)

bool operator<(thread::id x, thread::id y) noexcept;

8 Returns: A value such that operator< is a total ordering as described in 28.7.

bool operator<=(thread::id x, thread::id y) noexcept;

9 Returns: !(y < x).

bool operator>(thread::id x, thread::id y) noexcept;

10 Returns: y < x.

bool operator>=(thread::id x, thread::id y) noexcept;

11 Returns: !(x < y).

template<class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& out, thread::id id);

12 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 shall have the same text representation and if x != y the thread::id objects shall have distinct text representations.

13 Returns: out.

template<> struct hash<thread::id>;

14 The specialization is enabled (23.14.15).
### 33.3.2.2 thread constructors

```cpp
thread() noexcept;
```

**Effects:** Constructs a `thread` object that does not represent a thread of execution.

**Postconditions:** `get_id() == id()`.

```cpp
template<class F, class... Args> explicit thread(F&& f, Args&&... args);
```

**Requires:** `F` and each `T_i` in `Args` shall satisfy the `MoveConstructible` requirements. `INVOKEDECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...)` (23.14.3) shall be a valid expression.

**Remarks:** This constructor shall not participate in overload resolution if `decay_t<F>` is the same type as `std::thread`.

**Effects:** Constructs an object of type `thread`. The new thread of execution executes `INVOKEDECAY_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 `INVOKEDECAY_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`.

**Postconditions:** `get_id() != id()`. `*this` represents the newly started thread.

**Throws:** `system_error` if unable to start the new thread.

**Error conditions:**

— `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

```cpp
thread(thread&& x) noexcept;
```

**Effects:** Constructs an object of type `thread` from `x`, and sets `x` to a default constructed state.

**Postconditions:** `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction.

### 33.3.2.3 thread destructor

```cpp
~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]

### 33.3.2.4 thread assignment

```cpp
thread& operator=(thread&& x) noexcept;
```

**Effects:** If `joinable()`, calls `terminate()`. Otherwise, assigns the state of `x` to `*this` and sets `x` to a default constructed state.

**Postconditions:** `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment.

**Returns:** `*this`.

### 33.3.2.5 thread members

```cpp
void swap(thread& x) noexcept;
```

**Effects:** Swaps the state of `*this` and `x`.

```cpp
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]

Postconditions: The thread represented by *this has completed. get_id() == id().

Throws: system_error when an exception is required (33.2.2).

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.

Postconditions: get_id() == id().

Throws: system_error when an exception is required (33.2.2).

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.

33.3.2.6 thread static members [thread.thread.static]

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.

33.3.2.7 thread specialized algorithms [thread.thread.algorithm]

void swap(thread& x, thread& y) noexcept;

Effects: As if by x.swap(y).

33.3.3 Namespace this_thread [thread.thread.this]

namespace std::this_thread {
    thread::id get_id() noexcept;
    void yield() noexcept;
    template<class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template<class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
}

thread::id this_thread::get_id() noexcept;

Returns: An object of type thread::id that uniquely identifies the current thread of execution. No other thread of execution 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 (33.2.4) specified by abs_time.

Synchronization: None.

Throws: Timeout-related exceptions (33.2.4).

template<class Rep, class Period>
void sleep_for(const chrono::duration<Rep, Period>& rel_time);

Effects: Blocks the calling thread for the relative timeout (33.2.4) specified by rel_time.

Synchronization: None.

Throws: Timeout-related exceptions (33.2.4).

33.4 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).

33.4.1 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; };
  struct try_to_lock_t { explicit try_to_lock_t() = default; };
  struct adopt_lock_t { explicit adopt_lock_t() = default; };

  inline constexpr defer_lock_t defer_lock {{ }};
  inline constexpr try_to_lock_t try_to_lock {{ }};
  inline constexpr adopt_lock_t adopt_lock {{ }};

  template<class Mutex> class lock_guard;
  template<class... MutexTypes> class scoped_lock;
  template<class Mutex> class unique_lock;

  template<class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;

  template<class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
  template<class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

  struct once_flag;

  template<class Callable, class... Args>
  void call_once(once_flag& flag, Callable&& func, Args&&... args);
}

33.4.2 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;
}

§ 33.4.2
33.4.3 Mutex requirements

33.4.3.1 In general

A mutex object facilitates protection against data races and allows safe synchronization of data between execution agents (33.2.5). 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.

33.4.3.2 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 meet the requirements set out in this subclause. In this description, m denotes an object of a mutex type.

The mutex types shall meet the Lockable requirements (33.2.5.3).

The mutex types shall be DefaultConstructible and Destructible. 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 \texttt{m.lock()} shall be well-formed and have the following semantics:

\begin{itemize}
  \item Requires: If \texttt{m} is of type \texttt{mutex}, \texttt{timed_mutex}, \texttt{shared_mutex}, or \texttt{shared_timed_mutex}, the calling thread does not own the mutex.
  \item Effects: Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.
  \item Postconditions: The calling thread owns the mutex.
  \item Return type: \texttt{void}.
  \item Synchronization: Prior \texttt{unlock()} operations on the same object shall synchronize with (6.8.2) this operation.
  \item Throws: \texttt{system_error} when an exception is required (33.2.2).
  \item Error conditions:
    \begin{itemize}
      \item operation_not_permitted — if the thread does not have the privilege to perform the operation.
      \item resource_deadlock_would_occur — if the implementation detects that a deadlock would occur.
    \end{itemize}
\end{itemize}

The expression \texttt{m.try_lock()} shall be well-formed and have the following semantics:

\begin{itemize}
  \item Requires: If \texttt{m} is of type \texttt{mutex}, \texttt{timed_mutex}, \texttt{shared_mutex}, or \texttt{shared_timed_mutex}, the calling thread does not own the mutex.
  \item Effects: Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and \texttt{try_lock()} immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [Note: This spurious failure is normally uncommon, but allows interesting implementations based on a simple compare and exchange (Clause 32). — end note] An implementation should ensure that \texttt{try_lock()} does not consistently return \texttt{false} in the absence of contending mutex acquisitions.
  \item Return type: \texttt{bool}.
  \item Returns: \texttt{true} if ownership of the mutex was obtained for the calling thread, otherwise \texttt{false}.
  \item Synchronization: If \texttt{try_lock()} returns \texttt{true}, prior \texttt{unlock()} operations on the same object synchronize with (6.8.2) this operation. [Note: Since \texttt{lock()} does not synchronize with a failed subsequent \texttt{try_—
lock(), the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures. —end note]

Throws: Nothing.

The expression `m.unlock()` shall be well-formed and have the following semantics:

Requires: The calling thread shall own the mutex.

Effects: Releases the calling thread’s ownership of the mutex.

Return type: void.

Synchronization: This operation synchronizes with (6.8.2) subsequent lock operations that obtain ownership on the same object.

Throws: Nothing.

### 33.4.3.2.1 Class mutex

```cpp
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 33.2.3

    native_handle_type native_handle(); // see 33.2.3
};
}
```

1 The class `mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

2 [Note: After a thread A has called `unlock()`, releasing a mutex, it is possible for another thread B to lock the same mutex, observe that it is no longer in use, unlock it, and destroy it, before thread A appears to have returned from its unlock call. Implementations are required to handle such scenarios correctly, as long as thread A doesn’t access the mutex after the unlock call returns. These cases typically occur when a reference-counted object contains a mutex that is used to protect the reference count. —end note]

3 The class `mutex` shall satisfy all of the mutex requirements (33.4.3). It shall be a standard-layout class (Clause 12).

4 [Note: A program may deadlock if the thread that owns a `mutex` object calls `lock()` on that object. If the implementation can detect the deadlock, a resource_deadlock_would_occur error condition 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.

### 33.4.3.2.2 Class recursive_mutex

```cpp
namespace std {

class recursive_mutex {
public:
    recursive_mutex();
    ~recursive_mutex();

    recursive_mutex(const recursive_mutex&) = delete;
    recursive_mutex& operator=(const recursive_mutex&) = delete;
};
}
```
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 (33.4.3). It shall be a standard-layout class (Clause 12).

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.

### 33.4.3.3 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` (23.17.5), and `abs_time` denotes an object of an instantiation of `time_point` (23.17.6).

The timed mutex types shall meet the `TimedLockable` requirements (33.2.5.4).

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 (33.2.4) 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 (33.2.4).

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 (33.2.4) 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.

Throws: Timeout-related exceptions (33.2.4).

### 33.4.3.3.1 Class timed_mutex

```cpp
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 33.2.3
        native_handle_type native_handle(); // see 33.2.3
    };
}
```

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 (33.4.3.3). It shall be a standard-layout class (Clause 12).

3. The behavior of a program is undefined if:
   - it destroys a `timed_mutex` object owned by any thread,
   - a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object, or
   - a thread terminates while owning a `timed_mutex` object.

### 33.4.3.3.2 Class recursive_timed_mutex

```cpp
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();
    };
}
```

§ 33.4.3.3.2
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 (33.4.3.3). It shall be a standard-layout class (Clause 12).

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:

- it destroys a `recursive_timed_mutex` object owned by any thread, or
- a thread terminates while owning a `recursive_timed_mutex` object.

### 33.4.3.4 Shared mutex types

The standard library types `shared_mutex` and `shared_timed_mutex` are shared mutex types. Shared mutex types shall meet the requirements of mutex types (33.4.3.2), and additionally shall meet the requirements set out below. In this description, `m` denotes an object of a shared mutex type.

In addition to the exclusive lock ownership mode specified in 33.4.3.2, 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. **Postconditions:** 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 (33.2.2).

**Error conditions:**

- `operation_not_permitted` — if the thread does not have the privilege to perform the operation.
- `resource_deadlock_would_occur` — if the implementation detects that a deadlock would occur.

The expression `m.unlock_shared()` 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`.
Synchronization: This operation synchronizes with (6.8.2) subsequent lock() operations that obtain ownership on the same object.

Throws: Nothing.

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.

33.4.3.4.1 Class shared_mutex

The class shared_mutex provides a non-recursive mutex with shared ownership semantics.

The class shared_mutex shall satisfy all of the shared mutex requirements (33.4.3.4). It shall be a standard-layout class (Clause 12).

The behavior of a program is undefined if:

1. it destroys a shared_mutex object owned by any thread,
2. a thread attempts to recursively gain any ownership of a shared_mutex, or
3. a thread terminates while possessing any ownership of a shared_mutex.

shared_mutex may be a synonym for shared_timed_mutex.

33.4.3.5 Shared timed mutex types

The standard library type shared_timed_mutex is a shared timed mutex type. Shared timed mutex types shall meet the requirements of timed mutex types (33.4.3.3), shared mutex types (33.4.3.4), and additionally shall meet the requirements set out below. In this description, m denotes an object of a shared timed mutex type, rel_type denotes an object of an instantiation of duration (23.17.5), and abs_time denotes an object of an instantiation of time_point (23.17.6).
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 (33.2.4) specified by `rel_time`. If the time specified by `rel_time` is less than or equal to `rel_time.zero()`, the function attempts to obtain ownership without blocking (as if by calling `try_lock_shared()`). The function shall return within the timeout specified by `rel_time` only if it has obtained shared ownership of the mutex object. [Note: As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note] If an exception is thrown then a shared lock shall not have been acquired for the current thread.

***Return type:** `bool`.

***Returns:** `true` if the shared lock was acquired, `false` otherwise.

***Synchronization:** If `try_lock_shared_for()` returns `true`, prior `unlock()` operations on the same object synchronize with (6.8.2) this operation.

***Throws:** Timeout-related exceptions (33.2.4).

The expression `m.try_lock_shared_until(abs_time)` shall be well-formed and have the following semantics:

***Requires:** The calling thread has no ownership of the mutex.

***Effects:** The function attempts to obtain shared ownership of the mutex. If `abs_time` has already passed, the function attempts to obtain shared ownership without blocking (as if by calling `try_lock_shared()`). The function shall return before the absolute timeout (33.2.4) specified by `abs_time` only if it has obtained shared ownership of the mutex object. [Note: As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — end note] If an exception is thrown then a shared lock shall not have been acquired for the current thread.

***Return type:** `bool`.

***Returns:** `true` if the shared lock was acquired, `false` otherwise.

***Synchronization:** If `try_lock_shared_until()` returns `true`, prior `unlock()` operations on the same object synchronize with (6.8.2) this operation.

***Throws:** Timeout-related exceptions (33.2.4).

33.4.3.5.1 Class `shared_timed_mutex` [thread.sharedtimedmutex.class]

```cpp
namespace std {
    class shared_timed_mutex {
        public:
            shared_timed_mutex();
            ~shared_timed_mutex();

            shared_timed_mutex(const shared_timed_mutex&) = delete;
            shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;

            // exclusive ownership
            void lock(); // blocking
            bool try_lock();
            template<class Rep, class Period>
                bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
            void unlock();

            // shared ownership
            void lock_shared(); // blocking
            bool try_lock_shared();
            template<class Rep, class Period>
                bool try_lock_shared_for(const chrono::duration<Rep, Period>& rel_time);
            template<class Clock, class Duration>
                bool try_lock_shared_until(const chrono::time_point<Clock, Duration>& abs_time);

33.4.3.5.1 1228
```
The class `shared_timed_mutex` provides a non-recursive mutex with shared ownership semantics.

The class `shared_timed_mutex` shall satisfy all of the shared timed mutex requirements (33.4.3.5). It shall be a standard-layout class (Clause 12).

The behavior of a program is undefined if:

(3.1) — it destroys a `shared_timed_mutex` object owned by any thread,
(3.2) — a thread attempts to recursively gain any ownership of a `shared_timed_mutex`, or
(3.3) — a thread terminates while possessing any ownership of a `shared_timed_mutex`.

### 33.4.4 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
    struct adopt_lock_t { };  // assume the calling thread has already
    // obtained mutex ownership and manage it

    inline constexpr defer_lock_t defer_lock { };  
    inline constexpr try_to_lock_t try_to_lock { };  
    inline constexpr adopt_lock_t adopt_lock { };
}
```

### 33.4.4.1 Class template lock_guard

An object of type `lock_guard` controls the ownership of a lockable object within a scope. A `lock_guard` object maintains ownership of a lockable object throughout the `lock_guard` object’s lifetime (6.6.3). 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 meet the `BasicLockable` requirements (33.2.5.2).

```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 meet the `BasicLockable` requirements (33.2.5.2).

```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`. 

§ 33.4.4.1
Effects: As if by \texttt{m.lock()}.

\texttt{lock\_guard(mutex\_type& m, adopt\_lock\_t)};

\textbf{Requires}: The calling thread owns the mutex \texttt{m}.

\textbf{Postconditions}: \texttt{&pm == &m}

\textbf{Throws}: Nothing.

\texttt{~lock\_guard();}

Effects: As if by \texttt{pm.unlock()}.

\subsection{Class template \texttt{scoped\_lock}}

namespace std {
    template<class... MutexTypes>
    class scoped_lock {
        public:
            using mutex_type = Mutex; // If MutexTypes... consists of the single type Mutex

            explicit scoped_lock(MutexTypes&... m);
            explicit scoped_lock(adopt_lock_t, MutexTypes&... m);
            ~scoped_lock();

            scoped_lock(const scoped_lock&); = delete,
            scoped_lock& operator=(const scoped_lock&); = delete;

        private:
            tuple<MutexTypes&...> pm; // exposition only
        }
    }

An object of type \texttt{scoped\_lock} controls the ownership of lockable objects within a scope. A \texttt{scoped\_lock} object maintains ownership of lockable objects throughout the \texttt{scoped\_lock} object’s lifetime (6.6.3). The behavior of a program is undefined if the lockable objects referenced by \texttt{pm} do not exist for the entire lifetime of the \texttt{scoped\_lock} object. When \texttt{sizeof...(MutexTypes)} is 1, the supplied \texttt{Mutex} type shall meet the \texttt{BasicLockable} requirements (33.2.5.2). Otherwise, each of the mutex types shall meet the \texttt{Lockable} requirements (33.2.5.3).

\texttt{explicit scoped_lock(MutexTypes&... m);}

\textbf{Requires}: If a \texttt{MutexTypes} type is not a recursive mutex, the calling thread does not own the corresponding mutex element of \texttt{m}.

\textbf{Effects}: Initializes \texttt{pm} with \texttt{tie(m...)}. Then if \texttt{sizeof...(MutexTypes)} is 0, no effects. Otherwise if \texttt{sizeof...(MutexTypes)} is 1, then \texttt{m.lock()}. Otherwise, \texttt{lock(m...)}.

\texttt{explicit scoped_lock(adopt\_lock\_t, MutexTypes&... m);}

\textbf{Requires}: The calling thread owns all the mutexes in \texttt{m}.

\textbf{Effects}: Initializes \texttt{pm} with \texttt{tie(m...)}.

\textbf{Throws}: Nothing.

\texttt{~scoped\_lock();}

\textbf{Effects}: For all \texttt{i} in \([0, \texttt{sizeof...(MutexTypes)})\), \texttt{get\langle i\rangle(pm).unlock()}.

\subsection{Class template \texttt{unique\_lock}}

namespace std {
    template<class Mutex>
    class unique_lock {
        public:
            using mutex_type = Mutex;


\S 33.4.4.3
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 meet the `BasicLockable` requirements (33.2.5.2).

2 [Note: `unique_lock<Mutex>` meets the `BasicLockable` requirements. If `Mutex` meets the `Lockable` requirements (33.2.5.3), `unique_lock<Mutex>` also meets the `Lockable` requirements; if `Mutex` meets the `TimedLockable` requirements (33.2.5.4), `unique_lock<Mutex>` also meets the `TimedLockable` requirements. —end note]

### 33.4.4.3.1 unique_lock constructors, destructor, and assignment

```cpp
unique_lock() noexcept;
explicit unique_lock(mutex_type& m);
unique_lock(mutex_type& m, defer_lock_t) noexcept;
unique_lock(mutex_type& m, try_to_lock_t);
unique_lock(mutex_type& m, adopt_lock_t);
```

```cpp
template<class Clock, class Duration>
unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
```

```cpp
template<class Rep, class Period>
unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

```cpp
~unique_lock();
```

```cpp
unique_lock(const unique_lock&) = delete;
unique_lock& operator=(const unique_lock&) = delete;
```

```cpp
unique_lock(unique_lock&& u) noexcept;
unique_lock& operator=(unique_lock&& u);
```

#### locking

```cpp
void lock();
bool try_lock();
```

```cpp
template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

```cpp
template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

```cpp
void unlock();
```

#### modifiers

```cpp
void swap(unique_lock& u) noexcept;
mutex_type* release() noexcept;
```

#### observers

```cpp
bool owns_lock() const noexcept;
explicit operator bool () const noexcept;
mutex_type* mutex() const noexcept;
```

```cpp
private:
mutex_type* pm; // exposition only
bool owns; // exposition only
};
```

```cpp
template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
```
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().
Postconditions: pm == addressof(m) and owns == true.

unique_lock(mutex_type& m, defer_lock_t) noexcept;
Effects: Constructs an object of type unique_lock.
Postconditions: pm == addressof(m) and owns == false.

unique_lock(mutex_type& m, try_to_lock_t);
Requires: The supplied Mutex type shall meet the Lockable requirements (33.2.5.3). 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().
Postconditions: pm == addressof(m) and owns == res, where res is the value returned by the call
to m.try_lock().

unique_lock(mutex_type& m, adopt_lock_t);
Requires: The calling thread owns the mutex.
Effects: Constructs an object of type unique_lock.
Postconditions: pm == addressof(m) and owns == true.
Throws: Nothing.

template<class Clock, class Duration>
unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The
supplied Mutex type shall meet the TimedLockable requirements (33.2.5.4).
Effects: Constructs an object of type unique_lock and calls m.try_lock_until(abs_time).
Postconditions: pm == addressof(m) and owns == res, where res is the value returned by the call
to m.try_lock_until(abs_time).

template<class Rep, class Period>
unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
Requires: If mutex_type is not a recursive mutex the calling thread does not own the mutex. The
supplied Mutex type shall meet the TimedLockable requirements (33.2.5.4).
Effects: Constructs an object of type unique_lock and calls m.try_lock_for(rel_time).
Postconditions: pm == addressof(m) and owns == res, where res is the value returned by the call
to m.try_lock_for(rel_time).

unique_lock(unique_lock&& u) noexcept;
Postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this
construction), u.pm == 0 and u.owns == false.

effects: If owns calls pm->unlock().

unique_lock& operator=(unique_lock&& u);
Effects: If owns calls pm->unlock().

postconditions: pm == u_p.pm and owns == u_p.owns (where u_p is the state of u just prior to this
construction), u.pm == 0 and u.owns == false.

[Note: With a recursive mutex it is possible for both *this and u to own the same mutex before the
assignment. In this case, *this will own the mutex after the assignment and u will not. — end note]

Throws: Nothing.

~unique_lock();
Effects: If owns calls pm->unlock().

§ 33.4.4.3.1
33.4.4.3.2 unique_lock locking

void lock();

Effects: As if by pm->lock().

Postconditions: owns == true.

Throws: Any exception thrown by pm->lock(). system_error when an exception is required (33.2.2).

Error conditions:
(4.1) — operation_not_permitted — if pm is nullptr.
(4.2) — resource_deadlock_would_occur — if on entry owns is true.

bool try_lock();

Requires: The supplied Mutex shall meet the Lockable requirements (33.2.5.3).

Effects: As if by pm->try_lock().

Returns: The value returned by the call to try_lock().

Postconditions: owns == res, where res is the value returned by the call to try_lock().

Throws: Any exception thrown by pm->try_lock(). system_error when an exception is required (33.2.2).

Error conditions:
(10.1) — operation_not_permitted — if pm is nullptr.
(10.2) — resource_deadlock_would_occur — if on entry owns is true.

template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

Requires: The supplied Mutex type shall meet the TimedLockable requirements (33.2.5.4).

Effects: As if by pm->try_lock_until(abs_time).

Returns: The value returned by the call to try_lock_until(abs_time).

Postconditions: owns == res, where res is the value returned by the call to try_lock_until(abs_time).

Throws: Any exception thrown by pm->try_lock_until(). system_error when an exception is required (33.2.2).

Error conditions:
(16.1) — operation_not_permitted — if pm is nullptr.
(16.2) — resource_deadlock_would_occur — if on entry owns is true.

template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

Requires: The supplied Mutex type shall meet the TimedLockable requirements (33.2.5.4).

Effects: As if by pm->try_lock_for(rel_time).

Returns: The value returned by the call to try_lock_until(rel_time).

Postconditions: owns == res, where res is the value returned by the call to try_lock_until(rel_time).

Throws: Any exception thrown by pm->try_lock_for(). system_error when an exception is required (33.2.2).

Error conditions:
(22.1) — operation_not_permitted — if pm is nullptr.
(22.2) — resource_deadlock_would_occur — if on entry owns is true.

void unlock();

Effects: As if by pm->unlock().

Postconditions: owns == false.
25  Throws: system_error when an exception is required (33.2.2).

26  Error conditions:

26.1  — operation_not_permitted — if on entry owns is false.

33.4.4.3.3 unique_lock modifiers

void swap(unique_lock& u) noexcept;
1  Effects: Swaps the data members of *this and u.

mutex_type* release() noexcept;
2  Returns: The previous value of pm.

3  Postconditions: pm == 0 and owns == false.

template<class Mutex>
void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y) noexcept;
4  Effects: As if by x.swap(y).

33.4.4.3.4 unique_lock observers

bool owns_lock() const noexcept;
1  Returns: owns.

explicit operator bool() const noexcept;
2  Returns: owns.

mutex_type *mutex() const noexcept;
3  Returns: pm.

33.4.4.4 Class template shared_lock

namespace std {
    template<class Mutex>
    class shared_lock {
    public:
        using mutex_type = Mutex;

        // 33.4.4.4.1, 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;

        // 33.4.4.4.2, 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();
    };
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 meet the shared mutex requirements (33.4.3.5).

[Note: `shared_lock<Mutex>` meets the `TimedLockable` requirements (33.2.5.4). — end note]

### 33.4.4.4.1 shared_lock constructors, destructor, and assignment  [thread.lock.shared.cons]

```cpp
shared_lock() noexcept;
```

**Effects:** Constructs an object of type `shared_lock`.

**Postconditions:** `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()`.

**Postconditions:** `pm == addressof(m)` and `owns == true`.

```cpp
shared_lock(mutex_type& m, defer_lock_t) noexcept;
```

**Effects:** Constructs an object of type `shared_lock`.

**Postconditions:** `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()`.

**Postconditions:** `pm == addressof(m)` and `owns == res` where `res` is the value returned by the call to `m.try_lock_shared()`.

```cpp
shared_lock(mutex_type& m, adopt_lock_t);
```

**Requires:** The calling thread has shared ownership of the mutex.

**Effects:** Constructs an object of type `shared_lock`.

**Postconditions:** `pm == addressof(m)` and `owns == true`.

```cpp
template<class Clock, class Duration>
shared_lock(mutex_type& m,
            const chrono::time_point<Clock, Duration>& abs_time);
```

**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)`.
Postconditions: \( pm == \) addressof(m) and \( owns == res \) where \( res \) is the value returned by the call to m.try_lock_shared_until(abs_time).

```cpp
template<class Rep, class Period>
shared_lock(mutex_type& m,
    const chrono::duration<Rep, Period>& rel_time);
```

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_for(rel_time).
Postconditions: \( pm == \) addressof(m) and \( owns == res \) where \( res \) is the value returned by the call to m.try_lock_shared_for(rel_time).

~shared_lock();
Effects: If \( owns \) calls \( pm->unlock_shared() \).
shared_lock(shared_lock&& sl) noexcept;
Postconditions: \( pm == sl_p.pm \) and \( owns == sl_p.owns \) (where \( sl_p \) is the state of \( sl \) just prior to this construction), \( sl.pm == nullptr \) and \( sl.owns == false \).

shared_lock& operator=(shared_lock&& sl) noexcept;
Effects: If \( owns \) calls \( pm->unlock_shared() \).
Postconditions: \( pm == sl_p.pm \) and \( owns == sl_p.owns \) (where \( sl_p \) is the state of \( sl \) just prior to this assignment), \( sl.pm == nullptr \) and \( sl.owns == false \).

33.4.4.4.2 shared_lock locking

```cpp
void lock();
Effects: As if by \( pm->lock_shared() \).
Postconditions: \( owns == true \).
Throws: Any exception thrown by \( pm->lock_shared() \). system_error when an exception is required (33.2.2).
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();
Effects: As if by \( pm->try_lock_shared() \).
Returns: The value returned by the call to \( pm->try_lock_shared() \).
Postconditions: \( owns == res \), where \( res \) is the value returned by the call to \( pm->try_lock_shared() \).
Throws: Any exception thrown by \( pm->try_lock_shared() \). system_error when an exception is required (33.2.2).
Error conditions:
(9.1) — operation_not_permitted — if \( pm \) is nullptr.
(9.2) — resource_deadlock_would_occur — if on entry \( owns \) is true.
```

```cpp
template<class Clock, class Duration>
bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
Effects: As if by \( pm->try_lock_shared_until(abs_time) \).
Returns: The value returned by the call to \( pm->try_lock_shared_until(abs_time) \).
Postconditions: \( owns == res \), where \( res \) is the value returned by the call to \( pm->try_lock_shared_until(abs_time) \).
Throws: Any exception thrown by \( pm->try_lock_shared_until(abs_time) \). system_error when an exception is required (33.2.2).
Error conditions:
template<class Rep, class Period>
bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);

Effects: As if by pm->try_lock_shared_for(rel_time).

Returns: The value returned by the call to pm->try_lock_shared_for(rel_time).

Postconditions: owns == res, where res is the value returned by the call to pm->try_lock_shared_for(rel_time).

Throws: Any exception thrown by pm->try_lock_shared_for(rel_time). system_error when an exception is required (33.2.2).

Error conditions:
— operation_not_permitted — if pm is nullptr.
— resource_deadlock_would_occur — if on entry owns is true.

void unlock();

Effects: As if by pm->unlock_shared().

Postconditions: owns == false.

Throws: system_error when an exception is required (33.2.2).

Error conditions:
— operation_not_permitted — if pm is nullptr.
— resource_deadlock_would_occur — if on entry owns is true.

33.4.4.4.3 shared_lock modifiers

void swap(shared_lock& sl) noexcept;

Effects: Swaps the data members of *this and sl.

mutex_type* release() noexcept;

Returns: The previous value of pm.

Postconditions: pm == nullptr and owns == false.

template<class Mutex>
void swap(shared_lock<Mutex>& x, shared_lock<Mutex>& y) noexcept;

Effects: As if by x.swap(y).

33.4.4.4 shared_lock observers

bool owns_lock() const noexcept;

Returns: owns.

explicit operator bool() const noexcept;

Returns: owns.

mutex_type* mutex() const noexcept;

Returns: pm.

33.4.5 Generic locking algorithms

template<class L1, class L2, class... L3> int try_lock(L1& l1, L2& l2, L3&...);

Requires: Each template parameter type shall meet the Lockable 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() shall be called for all prior arguments and there shall be 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 meet the Lockable 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 shall 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() shall be called for any argument that had been locked by a call to lock() or try_lock().

33.4.6 Call once

33.4.6.1 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.

constexpr once_flag() noexcept;

Effects: Constructs an object of type once_flag.

Synchronization: The construction of a once_flag object is not synchronized.

Postconditions: The object’s internal state is set to indicate to an invocation of call_once with the object as its initial argument that no function has been called.

33.4.6.2 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 23.14.3) 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 (33.2.2), or any exception thrown by func.

[Example:

// global flag, regular function
void init();
std::once_flag flag;]
© ISO/IEC

33.5 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 maximum efficiency on some platforms. 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.

33.5.1 Header `<condition_variable>` synopsis

namespace std {
    class condition_variable;
    class condition_variable_any;

    void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

    enum class cv_status { no_timeout, timeout };
}

33.5.2 Non-member functions

void notify_all_at_thread_exit(condition_variable& cond, unique_lock<mutex> lk);

1 Requires: `lk` is locked by the calling thread and either

(1.1) — no other thread is waiting on `cond`, or
lk.mutex() returns the same value for each of the lock arguments supplied by all concurrently waiting (via wait, wait_for, or wait_until) threads.

**Effects:** Transfers ownership of the lock associated with lk into internal storage and schedules cond to be notified when the current thread exits, after all objects of thread storage duration associated with the current thread have been destroyed. This notification shall be as if:

```cpp
lk.unlock();
cond.notify_all();
```

**Synchronization:** The implied lk.unlock() call is sequenced after the destruction of all objects with thread storage duration associated with the current thread.

**[Note:** The supplied lock will be held until the thread exits, and care should be taken to ensure that this does not cause deadlock due to lock ordering issues. After calling `notify_all_at_thread_exit` it is recommended that the thread should be exited as soon as possible, and that no blocking or time-consuming tasks are run on that thread. —*end note*]

**[Note:** It is the user’s responsibility to ensure that waiting threads do not erroneously assume that the thread has finished if they experience spurious wakeups. This typically requires that the condition being waited for is satisfied while holding the lock on lk, and that this lock is not released and reacquired prior to calling `notify_all_at_thread_exit`. —*end note*]

### 33.5.3 Class condition_variable

```cpp
namespace std {
    class condition_variable {
        public:
            condition_variable();
            condition_variable(const condition_variable&) = delete;
            condition_variable& operator=(const condition_variable&) = delete;

            void notify_one() noexcept;
            void notify_all() noexcept;
            void wait(unique_lock<mutex>& lock);
            template<class Predicate>
                void wait(unique_lock<mutex>& lock, Predicate pred);
            template<class Clock, class Duration>
                cv_status wait_until(unique_lock<mutex>& lock,
                                    const chrono::time_point<Clock, Duration>& abs_time);
            template<class Clock, class Duration, class Predicate>
                bool wait_until(unique_lock<mutex>& lock,
                                 const chrono::time_point<Clock, Duration>& abs_time,
                                 Predicate pred);
            template<class Rep, class Period>
                cv_status wait_for(unique_lock<mutex>& lock,
                                    const chrono::duration<Rep, Period>& rel_time);
            template<class Rep, class Period, class Predicate>
                bool wait_for(unique_lock<mutex>& lock,
                              const chrono::duration<Rep, Period>& rel_time,
                              Predicate pred);

            using native_handle_type = implementation-defined;  // see 33.2.3
            native_handle_type native_handle();  // see 33.2.3
        }
    }
```

1 The class condition_variable shall be a standard-layout class (Clause 12).

condition_variable();

**Effects:** Constructs an object of type condition_variable.

**Throws:** `system_error` when an exception is required (33.2.2).

**Error conditions:**
— resource_unavailable_try_again — if some non-memory resource limitation prevents initialization.

~condition_variable();

**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.

void wait(unique_lock<mutex>& lock);

**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.

**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.

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (18.5.1). **Note:** This can happen if the re-locking of the mutex throws an exception. **— end note**

**Postconditions:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

**Throws:** Nothing.

template<class Predicate>  
void wait(unique_lock<mutex>& lock, Predicate pred);

**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.

**Effects:** Equivalent to:
while (!pred())
    wait(lock);

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (18.5.1). **Note:** This can happen if the re-locking of the mutex throws an exception. **— end note**

**Postconditions:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

**Throws:** Any exception thrown by pred.

template<class Clock, class Duration>  
cv_status wait_until(unique_lock<mutex>& lock,  
const chrono::time_point<Clock, Duration>& abs_time);

**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 (33.2.4) 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 (18.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Postconditions:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

**Returns:** cv_status::timeout if the absolute timeout (33.2.4) specified by abs_time expired, otherwise cv_status::no_timeout.

**Throws:** Timeout-related exceptions (33.2.4).

```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:

```
return wait_until(lock, chrono::steady_clock::now() + rel_time);
```

**Returns:** cv_status::timeout if the relative timeout (33.2.4) specified by rel_time expired, otherwise cv_status::no_timeout.

**Remarks:** If the function fails to meet the postcondition, terminate() shall be called (18.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Postconditions:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

**Throws:** Timeout-related exceptions (33.2.4).

```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:

```
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 (18.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

**Postconditions:** lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: The returned value indicates whether the predicate evaluated to true regardless of whether the timeout was triggered. — end note]

**Throws:** Timeout-related exceptions (33.2.4) or any exception thrown by pred.
template<class Rep, class Period, class Predicate>
bool wait_for(unique_lock<mutex>& lock,
    const chrono::duration<Rep, Period>& rel_time,
    Predicate pred);

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
(37.1)
— 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 (18.5.1). [Note:
This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: lock.owns_lock() is true and lock.mutex() is locked by the calling thread.

[Note: The returned value indicates whether the predicate evaluates to true regardless of whether the
timeout was triggered. — end note]

Throws: Timeout-related exceptions (33.2.4) or any exception thrown by pred.

33.5.4 Class condition_variable_any [thread.condition.condvarany]

A Lock type shall meet the BasicLockable requirements (33.2.5.2). [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 (33.2.2).
Error conditions:

- `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.
- `operation_not_permitted` — if the thread does not have the privilege to perform the operation.

```cpp
~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.

```cpp
void notify_one() noexcept;
```

Effects: If any threads are blocked waiting for `*this`, unblocks one of those threads.

```cpp
void notify_all() noexcept;
```

Effects: Unblocks all threads that are blocked waiting for `*this`.

```cpp
template<class Lock>
void wait(Lock& lock);
```

Effects:

1. Atomically calls `lock.unlock()` and blocks on `*this`.
2. When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
3. The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, or spuriously.

Remarks: If the function fails to meet the postcondition, `terminate()` shall be called (18.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: `lock` is locked by the calling thread.

Throws: Nothing.

```cpp
template<class Lock, class Predicate>
void wait(Lock& lock, Predicate pred);
```

Effects: Equivalent to:

```cpp
while (!pred())
    wait(lock);
```

```cpp
template<class Lock, class Clock, class Duration>
cv_status wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
```

Effects:

1. Atomically calls `lock.unlock()` and blocks on `*this`.
2. When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
3. The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, expiration of the absolute timeout (33.2.4) specified by `abs_time`, or spuriously.
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 (18.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]

Postconditions: `lock` is locked by the calling thread.

Returns: `cv_status::timeout` if the absolute timeout (33.2.4) specified by `abs_time` expired, otherwise `cv_status::no_timeout`.

Throws: Timeout-related exceptions (33.2.4).
template<class Lock, class Rep, class Period>
    cv_status wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
19
    Effects: Equivalent to:
            return wait_until(lock, chrono::steady_clock::now() + rel_time);
20
    Returns: cv_status::timeout if the relative timeout (33.2.4) specified by rel_time expired, otherwise
            cv_status::no_timeout.
21
    Remarks: If the function fails to meet the postcondition, terminate() shall be called (18.5.1). [Note: This can happen if the re-locking of the mutex throws an exception. — end note]
22
    Postconditions: lock is locked by the calling thread.
23
    Throws: Timeout-related exceptions (33.2.4).

template<class Lock, class Clock, class Duration, class Predicate>
    bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
24
    Effects: Equivalent to:
            while (!pred())
                if (wait_until(lock, abs_time) == cv_status::timeout)
                    return pred();
            return true;
25
    [Note: There is no blocking if pred() is initially true, or if the timeout has already expired. — end note]
26
    [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);
27
    Effects: Equivalent to:
            return wait_until(lock, chrono::steady_clock::now() + rel_time, std::move(pred));

33.6 Futures

33.6.1 Overview

33.6 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]

33.6.2 Header <future> synopsis

namespace std {
    enum class future_errc {
        broken_promise = implementation-defined,
        future_already_retrieved = implementation-defined,
        promise_already_satisfied = implementation-defined,
        no_state = implementation-defined
    };
}

enum class launch : unspecified {
    async = unspecified,
    deferred = unspecified,
    implementation-defined
};

enum class future_status {
    ready,
    timeout,
    deferred
};
template<> struct is_error_code_enum<future_errc> : public true_type { }
error_code make_error_code(future_errc e) noexcept;
error_condition make_error_condition(future_errc e) noexcept;

const error_category& future_category() noexcept;

class future_error;

template<class R> class promise;
template<class R> class promise<R&>;
template<> class promise<void>;

template<class R>
void swap(promise<R>& x, promise<R>& y) noexcept;

template<class R, class Alloc>
struct uses_allocator<promise<R>, Alloc>;

template<class R> class future;
template<class R> class future<R&>;
template<> class future<void>;

template<class R> class shared_future;
template<class R> class shared_future<R&>;
template<> class shared_future<void>;

template<class> class packaged_task; // not defined

template<class R, class... ArgTypes>
class packaged_task<R(ArgTypes...)>;  // not defined

template<class R, class... ArgTypes>
void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

§ 33.6.3 1246
### 33.6.4 Class `future_error`

```cpp
namespace std {
    class future_error : public logic_error {
    public:
        explicit future_error(future_errc e);

        const error_code& code() const noexcept;
        const char* what() const noexcept;

    private:
        error_code ec_; // exposition only
    }
}
```

1. **Effects:** Constructs an object of class `future_error` and initializes `ec_` with `make_error_code(e)`.
2. **Returns:** `ec_`.
3. **Returns:** An NTBS incorporating `code().message()`.

### 33.6.5 Shared 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 (33.2.5), 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. if the 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 (6.6.4.2) 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 (20.5.5.9). For example, concurrent accesses through references returned by shared_future::get() (33.6.8) must either use read-only operations or provide additional synchronization. —end note]

### 33.6.6 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>;
    }
```

1 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.

2 The set_value, set_exception, set_value_at_thread_exit, and set_exception_at_thread_exit member functions behave as though they acquire a single mutex associated with the promise object while updating the promise object.
: true_type { };  

Requires: Alloc shall be an Allocator (20.5.3.5).

promise();

template<class Allocator>
    promise(allocator_arg_t, const Allocator& a);

    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;

    Effects: Constructs a new promise object and transfers ownership of the shared state of rhs (if any) to the newly-constructed object.

    Postconditions: rhs has no shared state.

~promise();

    Effects: Abandons any shared state (33.6.5).

promise& operator=(promise&& rhs) noexcept;

    Effects: Abandons any shared state (33.6.5) and then as if promise(std::move(rhs)).swap(*this).

    Returns: *this.

void swap(promise& other) noexcept;

    Effects: Exchanges the shared state of *this and other.

    Postconditions: *this has the shared state (if any) that other had prior to the call to swap. other has the shared state (if any) that *this had prior to the call to swap.

future<R> get_future();

    Returns: A future<R> object with the same shared state as *this.

    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:
(14.1) — future_already_retrieved if get_future has already been called on a promise with the same shared state as *this.
(14.2) — no_state if *this has no shared state.

void promise::set_value(const R& r);
void promise::set_value(R&& r);
void promise<R&>::set_value(R& r);
void promise<void>::set_value();

    Effects: Atomically stores the value r in the shared state and makes that state ready (33.6.5).

    Throws:
(16.1) — future_error if its shared state already has a stored value or exception, or
(16.2) — for the first version, any exception thrown by the constructor selected to copy an object of R, or
(16.3) — for the second version, any exception thrown by the constructor selected to move an object of R.

Error conditions:
(17.1) — promise_already_satisfied if its shared state already has a stored value or exception.
(17.2) — no_state if *this has no shared state.

void set_exception(exception_ptr p);

    Requires: p is not null.

    Effects: Atomically stores the exception pointer p in the shared state and makes that state ready (33.6.5).

    Throws: future_error if its shared state already has a stored value or exception.

Error conditions:
(21.1) promise_already_satisfied if its shared state already has a stored value or exception.

(21.2) no_state if *this has no shared state.

```cpp
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:

- future_error if its shared state already has a stored value or exception, or
- for the first version, any exception thrown by the constructor selected to copy an object of R, or
- for the second version, any exception thrown by the constructor selected to move an object of R.

Error conditions:

- promise_already_satisfied if its shared state already has a stored value or exception.
- no_state if *this has no shared state.

```cpp
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:

- promise_already_satisfied if its shared state already has a stored value or exception.
- no_state if *this has no shared state.

```cpp
template<class R>
void swap(promise<R>& x, promise<R>& y) noexcept;
```

Effects: As if by x.swap(y).

### 33.6.7 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 (33.6.5) 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.

1. [Note: Member functions of future do not synchronize with themselves or with member functions of shared_future. — end note]

The effect of calling any member function other than the destructor, the move-assignment operator, share, or valid on a future object for which valid() == false is undefined. [Note: It is valid to move from a future object for which valid() == false. — end note] [Note: Implementations should detect this case and throw an object of type future_error with an error condition of future_errc::no_state. — end note]

```cpp
namespace std {
    template<class R>
    class future {
        public:
            future() noexcept;
            future(future&&) noexcept;
            future(const future& rhs) = delete;
            ~future();
    }
}
```
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.

```cpp
future() noexcept;
Effects: Constructs an empty future object that does not refer to a shared state.
Postconditions: valid() == false.
```

```cpp
future(future&& rhs) noexcept;
Effects: Move constructs a future object that refers to the shared state that was originally referred to by rhs (if any).
Postconditions:
(8.1) valid() returns the same value as rhs.valid() prior to the constructor invocation.
(8.2) rhs.valid() == false.

~future();
Effects:
(9.1) Releases any shared state (33.6.5);
(9.2) destroys*this.
```

```cpp
future& operator=(future&& rhs) noexcept;
Effects:
(10.1) Releases any shared state (33.6.5).
(10.2) move assigns the contents of rhs to *this.
Postconditions:
(11.1) valid() returns the same value as rhs.valid() prior to the assignment.
(11.2) rhs.valid() == false.
```

```cpp
shared_future<R> share() noexcept;
Returns: shared_future<R>(std::move(*this)).
Postconditions: valid() == false.
```

```cpp
R future::get();
R& future<R&>::get();
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:
```
wait()s until the shared state is ready, then retrieves the value stored in the shared state;
— releases any shared state (33.6.5).

Returns:
— future::get() returns the value v stored in the object’s shared state as std::move(v).
— future<R&>::get() returns the reference stored as value in the object’s shared state.
— future<void>::get() returns nothing.

Throws: The stored exception, if an exception was stored in the shared state.

Postconditions: valid() == false.

bool valid() const noexcept;

Returns: true only if *this refers to a shared state.

void wait() const;

Effects: Blocks until the shared state is ready.

template<class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;

Effects: None if the shared state contains a deferred function (33.6.9), otherwise blocks until the shared state is ready or until the relative timeout (33.2.4) specified by rel_time has expired.

Returns:
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the relative timeout (33.2.4) specified by rel_time has expired.

Throws: timeout-related exceptions (33.2.4).

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 (33.6.9), otherwise blocks until the shared state is ready or until the absolute timeout (33.2.4) specified by abs_time has expired.

Returns:
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the absolute timeout (33.2.4) specified by abs_time has expired.

Throws: timeout-related exceptions (33.2.4).

33.6.8 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 (33.6.5) 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]

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.
6 Postconditions: valid() == false.

shared_future(const shared_future& rhs) noexcept;
7 Effects: Constructs a shared_future object that refers to the same shared state as rhs (if any).
8 Postconditions: valid() returns the same value as rhs.valid().

shared_future(future<R>&& rhs) noexcept;
shared_future(shared_future&& rhs) noexcept;
9 Effects: Move constructs a shared_future object that refers to the shared state that was originally referred to by rhs (if any).
10 Postconditions:
(10.1) — valid() returns the same value as rhs.valid() returned prior to the constructor invocation.
(10.2) — rhs.valid() == false.

~shared_future();
11 Effects:
(11.1) — Releases any shared state (33.6.5);
(11.2) — destroys *this.

shared_future& operator=(shared_future&& rhs) noexcept;
12 Effects:
(12.1) — Releases any shared state (33.6.5);
(12.2) — move assigns the contents of rhs to *this.
13 Postconditions:
(13.1) — valid() returns the same value as rhs.valid() returned prior to the assignment.
shared_future& operator=(const shared_future& rhs) noexcept;

Effects:
— Releases any shared state (33.6.5);
— Assigns the contents of rhs to *this. [Note: As a result, *this refers to the same shared state as rhs (if any). — end note]

Postconditions: valid() == rhs.valid().

const R& shared_future::get() const;
R& shared_future<R&>::get() const;
void shared_future<void>::get() const;

[Note: As described above, the template and its two required specializations differ only in the return type and return value of the member function get. — end note]

[Note: Access to a value object stored in the shared state is unsynchronized, so programmers should apply only those operations on R that do not introduce a data race (6.8.2). — end note]

Effects: wait() until the shared state is ready, then retrieves the value stored in the shared state.

Returns:
— shared_future::get() returns a const reference to the value stored in the object’s shared state.
  [Note: Access through that reference after the shared state has been destroyed produces undefined behavior; this can be avoided by not storing the reference in any storage with a greater lifetime than the shared_future object that returned the reference. — end note]
— shared_future<R&>::get() returns the reference stored as value in the object’s shared state.
— shared_future<void>::get() returns nothing.

Throws: The stored exception, if an exception was stored in the shared state.

bool valid() const noexcept;

Returns: true only if *this refers to a shared state.

void wait() const;

Effects: Blocks until the shared state is ready.

template<class Rep, class Period>
future_status wait_for(const chrono::duration<Rep, Period>& rel_time) const;

Effects: None if the shared state contains a deferred function (33.6.9), otherwise blocks until the shared state is ready or until the relative timeout (33.2.4) specified by rel_time has expired.

Returns:
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the relative timeout (33.2.4) specified by rel_time has expired.

Throws: timeout-related exceptions (33.2.4).

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 (33.6.9), otherwise blocks until the shared state is ready or until the absolute timeout (33.2.4) specified by abs_time has expired.

Returns:
— future_status::deferred if the shared state contains a deferred function.
— future_status::ready if the shared state is ready.
— future_status::timeout if the function is returning because the absolute timeout (33.2.4) specified by abs_time has expired.
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.

```cpp
template<class F, class... Args>
[nodiscard] future<invoke_result_t<decay_t<F>, decay_t<Args>...>>
async(F&& f, Args&&... args);
```

**Requires:** `F` and each `T_i` in `Args` shall satisfy the `MoveConstructible` requirements, and

```cpp
INVOKE(DECAY_COPY(std::forward<F>(f)),
    DECAY_COPY(std::forward<Args>(args))...) // see 23.14.3, 33.3.2.2
```

shall be a valid expression.

**Effects:** The first function behaves the same as a call to the second function with a policy argument of

- `launch::async` | `launch::deferred` and the same arguments for `F` and `Args`. The second function creates a shared state that is associated with the returned `future` object. The further behavior of the second function depends on the policy argument as follows (if more than one of these conditions applies, the implementation may choose any of the corresponding policies):

  1. If `launch::async` is set in policy, calls `INVOKE(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...)` (23.14.3, 33.3.2.2) 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(DECAY_COPY(std::forward<F>(f)), DECAY_COPY(std::forward<Args>(args))...)` is stored as the exceptional result in the shared state. The `thread` object is stored in the shared state and affects the behavior of any asynchronous return objects that reference that state.

  2. If `launch::deferred` is set in policy, stores `DECAY_COPY(std::forward<F>(f))` and `DECAY_COPY(std::forward<Args>(args))...` in the shared state. These copies of `f` and `args` constitute a `deferred function`. Invocation of the deferred function evaluates `INVOKE(std::move(g), std::move(xyz))` where `g` is the stored value of `DECAY_COPY(std::forward<F>(f))` and `xyz` is the stored copy of `DECAY_COPY(std::forward<Args>(args))...`. Any return value is stored as the result in the shared state. Any exception propagated from the execution of the deferred function is stored as the exceptional result in the shared state. The shared state is not made ready until the function has completed. The first call to a non-timed waiting function (33.6.5) 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(std::move(g), std::move(xyz))` begins, the function is no longer considered deferred. [Note: If this policy is specified together with other policies, such as when using a policy value of `launch::async` | `launch::deferred`, implementations should defer invocation or the selection of the policy when no more concurrency can be effectively exploited. — end note]

  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,

1. 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
2. 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,

(5.3) — a call to a waiting function on an asynchronous return object that shares the shared state created by this `async` call shall block until the associated thread has completed, as if joined, or else time out (33.3.2.5);

(5.4) — 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:*

(7.1) — `resource_unavailable_try_again` — if `policy == launch::async` and the system is unable to start a new thread.

8 [Example:]

```cpp
int work1(int value);  
int work2(int value);  
int work(int value) {  
  auto handle = std::async([=]{ return work2(value); });  
  int tmp = work1(value);  
  return tmp + handle.get();  // #1  
}
```

[Note: Line #1 might not result in concurrency because the `async` call uses the default policy, which may use `launch::deferred`, in which case the lambda might not be invoked until the `get()` call; in that case, `work1` and `work2` are called on the same thread and there is no concurrency. — end note] — end example]

33.6.10 Class template `packaged_task` [futures.task]

1 The class template `packaged_task` defines a type for wrapping a function or callable object so that the return value of the function or callable object is stored in a future when it is invoked.

2 When the `packaged_task` object is invoked, its stored task is invoked and the result (whether normal or exceptional) stored in the shared state. Any futures that share the shared state will then be able to access the stored result.

```cpp
namespace std {  
  template<class> class packaged_task; // not defined

  template<class R, class... ArgTypes>
  class packaged_task<R(ArgTypes...)> {  
    public:
      // construction and destruction  
      packaged_task() noexcept;  
      template<class F>  
      explicit packaged_task(F&& f);  
      ~packaged_task();

      // no copy  
      packaged_task(const packaged_task&) = delete;  
      packaged_task& operator=(const packaged_task&) = delete;

      // move support  
      packaged_task(packaged_task&& rhs) noexcept;  
      packaged_task& operator=(packaged_task&& rhs) noexcept;  
      void swap(packaged_task& other) noexcept;

      bool valid() const noexcept;

      // result retrieval  
      future<R> get_future();

      // execution  
      void operator()(ArgTypes...);
  }
```

§ 33.6.10 1256
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;

33.6.10.1 packaged_task 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, t_1, t_2, ..., t_N) (23.14.3), where t_1, t_2, ..., t_N 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 decay_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.

Postconditions: rhs has no shared state.

packaged_task& operator=(packaged_task&& rhs) noexcept;

Effects:
— (8.1) Releases any shared state (33.6.5);
— (8.2) calls packaged_task(std::move(rhs)).swap(*this).

~packaged_task();

Effects: Abandons any shared state (33.6.5).

void swap(packaged_task& other) noexcept;

Effects: Exchanges the shared states and stored tasks of *this and other.

Postconditions: *this has the same shared state and stored task (if any) as other prior to the call to swap. other has the same shared state and stored task (if any) as *this prior to the call to swap.

bool valid() const noexcept;

Returns: true only if *this has a shared state.

future<R> get_future();

Returns: A future object that shares the same shared state as *this.

Throws: A future_error object if an error occurs.

Error conditions:
— future_already_retrieved if get_future has already been called on a packaged_task object with the same shared state as *this.
— no_state if *this has no shared state.
void operator()(ArgTypes... args);

Effects: As if by `INVOKE<R>(f, t_1, t_2, ..., t_N)` (23.14.3), where `f` is the stored task of `*this` and `t_1, t_2, ..., t_N` are the values in `args`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. The shared state of `*this` is made ready, and any threads blocked in a function waiting for the shared state of `*this` to become ready are unblocked.

Throws: A `future_error` exception object if there is no shared state or the stored task has already been invoked.

Error conditions:

(18.1) — promise_already_satisfied if the stored task has already been invoked.
(18.2) — no_state if `*this` has no shared state.

void make_ready_at_thread_exit(ArgTypes... args);

Effects: As if by `INVOKE<R>(f, t_1, t_2, ..., t_N)` (23.14.3), where `f` is the stored task and `t_1, t_2, ..., t_N` are the values in `args`. If the task returns normally, the return value is stored as the asynchronous result in the shared state of `*this`, otherwise the exception thrown by the task is stored. In either case, this shall be done without making that state ready (33.6.5) 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:

(21.1) — promise_already_satisfied if the stored task has already been invoked.
(21.2) — no_state if `*this` has no shared state.

void reset();

Effects: As if `*this = packaged_task(std::move(f))`, where `f` is the task stored in `*this`. [Note: This constructs a new shared state for `*this`. The old state is abandoned (33.6.5). — end note]

Throws:

(23.1) — bad_alloc if memory for the new shared state could not be allocated.
(23.2) — any exception thrown by the move constructor of the task stored in the shared state.
(23.3) — future_error with an error condition of no_state if `*this` has no shared state.

33.6.10.2 packaged_task globals [futures.task.nonmembers]

```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)`. 

§ 33.6.10.2 1258
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 (9.8, 10.1, 13.2) 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 (10.1.3), namespace (10.3.1), class (Clause 12), enumeration (10.2), and template (Clause 17) declarations.

typedef-name:
    identifier

namespace-name:
    identifier
    namespace-alias

namespace-alias:
    identifier

class-name:
    identifier
    simple-template-id

enum-name:
    identifier

template-name:
    identifier

Note that a typedef-name naming a class is also a class-name (12.1).

A.2 Lexical conventions

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

universal-character-name:
    \a 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 "

§ A.2
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 ? :: . .* -> ->*
  ! + - * / % ^ & |
  = += -- *= /= %= ^= &= |=
  == !== >= <= >= <=<> &<& & &|<< >> >>= <<= <=> &<< &< &
  <<= <<= >>= + + -- ,
  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-suffix_opt
  octal-literal integer-suffix_opt
  decimal-literal integer-suffix_opt
  hexadecimal-literal integer-suffix_opt

§ A.2 1260
binary-literal:
  0b binary-digit
  0B binary-digit
  binary-literal 'opt binary-digit

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

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

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

binary-digit:
  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

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
   \' \" \? \\ \a \b \f \n \r \t \v
octal-escape-sequence:
   \ octal-digit
   \ octal-digit octal-digit
   \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
   \x hexadecimal-digit
   hexadecimal-escape-sequence hexadecimal-digit
floating-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}
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 \ sign_{opt} digit
floating-suffix: one of
   f l 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 Basic concepts [gram.basic]

translation-unit:
    declaration-seq_opt

A.4 Expressions [gram.expr]

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
  ~
  class-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 ...opt
  capture-list , capture ...opt
capture:
  simple-capture
  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
  -> cv-qualifier-seq_opt constrained-parameter cv-qualifier-seq_opt abstract-declarator_opt

nested-requirement:
  requires constraint-expression ;

postfix-expression:
  primary-expression
  postfix-expression [ expr-or-braced-init-list ]
  postfix-expression ( expression-list_opt )
  simple-type-specifier ( expression-list_opt )
  typename-specifier ( expression-list_opt )
  simple-type-specifier braced-init-list
  typename-specifier braced-init-list
  postfix-expression . template_opt id-expression
  postfix-expression -> template_opt id-expression
  postfix-expression . pseudo-destructor-name
  postfix-expression -> pseudo-destructor-name
  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

pseudo-destructor-name:
  nested-name-specifier_opt type-name :: ~ type-name
  nested-name-specifier template simple-template-id :: ~ type-name
  ~ type-name
  ~ decltype-specifier
unary-expression:
  postfix-expression
  ++ cast-expression
  -- cast-expression
  unary-operator cast-expression
  sizeof unary-expression
  sizeof ( type-id )
  sizeof ... ( identifier )
  alignof ( type-id )
  noexcept-expression
  new-expression
  delete-expression

unary-operator: one of
  * & + - ! ~

new-expression:
  :: opt new new-placementopt new-type-id new-initializeropt
  :: opt new new-placementopt ( type-id ) new-initializeropt

new-placement:
  ( expression-list )

new-type-id:
  type-specifier-seq new-declaratoropt

new-declarator:
  ptr-operator new-declaratoropt
  noptr-new-declarator

noptr-new-declarator:
  [ expression ] attribute-specifier-seqopt
  noptr-new-declarator [ constant-expression ] attribute-specifier-seqopt

new-initializer:
  ( expression-listopt )
  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

conditional-expression:
  logical-or-expression
  conditional-expression ? expression : assignment-expression

throw-expression:
  throw assignment-expression

assignment-expression:
  conditional-expression
  logical-or-expression assignment-operator initializer-clause
  throw-expression

assignment-operator: one of
  = *= /= %= += -= >>= <<= &= ^= |=

equation:
  assignment-expression
  expression , assignment-expression

constant-expression:
  conditional-expression

A.5 Statements [gram.stmt]

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 ;
  goto identifier ;

declaration-statement:
  block-declaration

A.6 Declarations
[gram.dcl]

declaration-seq:
  declaration
  declaration-seq declaration

declaration:
  block-declaration
  nodeclspec-function-declaration
  function-definition
  template-declaration
  deduction-guide
  explicit-instantiation
  explicit-specialization
  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-seq_opt declarator ;

alias-declaration:
    using identifier attribute-specifier-seq_opt = defining-type-id ;

simple-declaration:
    decl-specifier-seq init-declarator-list_opt ;
    attribute-specifier-seq decl-specifier-seq init-declarator-list ;
    attribute-specifier-seq_opt decl-specifier-seq ref-qualifier_opt [ identifier-list ] initializer ;

static_assert-declaration:
    static_assert ( constant-expression ) ;
    static_assert ( constant-expression , string-literal ) ;

empty-declaration:

attribute-declaration:
    attribute-specifier-seq ;

decl-specifier:
    storage-class-specifier
    defining-type-specifier
    function-specifier
    friend
    typedef
    constexpr
    inline

decl-specifier-seq:
    decl-specifier attribute-specifier-seq_opt
decl-specifier decl-specifier-seq

storage-class-specifier:
    static
    thread_local
    extern
    mutable

function-specifier:
    virtual
    explicit
typedef-name:
    identifier
type-specifier:
    simple-type-specifier
    elaborated-type-specifier
    typename-specifier
    cv-qualifier
type-specifier-seq:
    type-specifier attribute-specifier-seq_opt
type-specifier type-specifier-seq
defining-type-specifier:
    type-specifier
class-specifier
defining-type-specifier

creating-type-specifier:
    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
  nested-name-specifier\opt template-name
  char
  char16_t
  char32_t
  wchar_t
  bool
  short
  int
  long
  signed
  unsigned
  float
  double
  void
  auto
  decltype-specifier

type-name:
  class-name
  enum-name
  typedef-name
  simple-template-id

decltype-specifier:
  decltype( expression )
  decltype( auto )

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

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 nested-name-specifier\opt identifier enum-base\opt ;

enum-key:
  enum
  enum class
  enum struct

enum-base:
  : type-specifier-seq

enumerator-list:
  enumerator-definition
  enumerator-list , enumerator-definition

enumerator-definition:
  enumerator
  enumerator = constant-expression

enumerator:
  identifier attribute-specifier-seq\opt
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 :: identifier { namespace-body }

enclosing-namespace-specifier:
    identifier
    enclosing-namespace-specifier :: identifier

namespace-body:
    declaration-seq_opt

namespace-alias:
    identifier

namespace-alias-definition:
    namespace identifier = qualified-namespace-specifier ;

qualified-namespace-specifier:
    namespace identifier_opt namespace-name

using-declaration:
    using using-declarator-list ;

using-declarator-list:
    using-declarator ...opt
    using-declarator-list , using-declarator ...opt

using-declarator:
    typename_opt nested-name-specifier unqualified-id

using-directive:
    attribute-specifier-seq_opt using namespace nested-name-specifier_opt namespace-name ;

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 ]]
    alignment-specifier

alignment-specifier:
    alignas ( type-id ...opt )
    alignas ( constant-expression ...opt )

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-namespaced :: identifier
attribute-namespaced:
  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

A.7  Declarators  [gram.decl]

init-declarator-list:
  init-declarator
  init-declarator-list , init-declarator
init-declarator:
  declarator initializeropt
  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-seqopt
  noptr-declarator parameters-and-qualifiers
  noptr-declarator [ constant-expressionopt ] attribute-specifier-seqopt
  ( ptr-declarator )
parameters-and-qualifiers:
  ( parameter-declaration-clause ) cv-qualifier-seqopt
  ref-qualifieropt noexcept-specifieropt attribute-specifier-seqopt
trailing-return-type:
  -> type-id
ptr-operator:
  * attribute-specifier-seqopt cv-qualifier-seqopt
  & attribute-specifier-seqopt
  && attribute-specifier-seqopt
  nested-name-specifier * attribute-specifier-seqopt cv-qualifier-seqopt
cv-qualifier-seq:
  cv-qualifier cv-qualifier-seqopt
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 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
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 ;
initializer:
  brace-or-equal-initializer
  ( expression-list )brace-or-equal-initializer:
  = initializer-clause
  braced-init-listinitializer-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

A.8 Classes

class-name:
identifier
simple-template-id
class-specifier:
class-head { member-specification_opt }
class-head:
class-key attribute-specifier-seq_opt class-head-name class-virt-specifier_opt base-clause_opt
class-key attribute-specifier-seq_opt base-clause_opt
class-head-name:
nested-name-specifier_opt class-name
class-virt-specifier:
final
class-key:
class
struct
union
member-specification:
member-declaration member-specification_opt
access-specifier : member-specification_opt
member-declaration:
attribute-specifier-seq_opt decl-specifier-seq_opt member-declarator-list_opt ;
function-definition
using-declaration
static_assert-declaration
template-declaration
deduction-guide
alias-declaration
empty-declaration
member-declarator-list:
member-declarator
member-declarator-list , member-declarator
member-declarator:
declarator virt-specifier-seq_opt pure-specifier_opt
declarator requires-clause
declarator brace-or-equal-initializer_opt
identifier_opt attribute-specifier-seq_opt : constant-expression brace-or-equal-initializer_opt
virt-specifier-seq:
virt-specifier
virt-specifier-seq virt-specifier
virt-specifier:
override
final
pure-specifier:
= 0
A.9 Derived classes

base-clause:
    : base-specifier-list

base-specifier-list:
    base-specifier ...opt
    base-specifier-list , base-specifier ...opt

base-specifier:
    attribute-specifier-seqopt class-or-decltype
    attribute-specifier-seqopt virtual access-specifieropt class-or-decltype
    attribute-specifier-seqopt access-specifier virtualopt class-or-decltype

class-or-decltype:
    nested-name-specifier opt class-name
    nested-name-specifier template simple-template-id
declay-type-specifier

access-specifier:
    private
    protected
    public

A.10 Special member functions

conversion-function-id:
    operator conversion-type-id

conversion-type-id:
    type-specifier-seq conversion-declaratoropt

conversion-declarator:
    ptr-operator conversion-declaratoropt

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

declay-type-specifier

A.11 Overloading

operator-function-id:
    operator operator

operator: one of
    new delete new[] delete[] () ] ] -> ->* ~
    ! + - * / % ^ & |   |
    = == -= -= /= %= ^= &= |= |
    == != < > <= >= <=> &<& &<& | |
    << >>= <= >= ++ -- , ,

literal-operator-id:
    operator string-literal identifier
    operator user-defined-string-literal

A.12 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

template-parameter:
  type-parameter
  parameter-declaration
  constrained-parameter
type-parameter:
  type-parameter-key ... opt identifier opt
  type-parameter-key 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
type-name

constrained-parameter:
  qualified-concept-name ... identifier opt
  qualified-concept-name identifier opt default-template-argument opt
qualified-concept-name:
  nested-name-specifier opt concept-name
  nested-name-specifier opt partial-concept-id

partial-concept-id:
  concept-name < template-argument-list opt >
default-template-argument:
  = type-id
  = id-expression
  = initializer-clause

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
typedef-name-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.13 Exception handling  [gram.except]

try-block:
  try compound-statement handler-seq

function-try-block:
  try ctor-initializer-opt compound-statement handler-seq

handler-seq:
  handler handler-seq-opt

handler:
  catch ( exception-declaration ) compound-statement

exception-declaration:
  attribute-specifier-seq_opt type-specifier-seq declarator
  attribute-specifier-seq_opt type-specifier-seq abstract-declarator_opt...
	noexcept-specifier:
  noexcept ( constant-expression )
  noexcept
  throw ( )

A.14 Preprocessing directives  [gram.cpp]

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 lparen identifier-list_opt ) replacement-list new-line
  # define identifier lparen ... ) replacement-list new-line
  # define identifier lparen identifier-list , ... ) replacement-list new-line
  # undef identifier new-line
  # line pp-tokens new-line
  # error pp-tokens new-line
  # pragma pp-tokens new-line
  # new-line

if-section:
  if-group elif-groups_opt else-group_opt endif-line

if-group:
  # if constant-expression new-line group_opt
  # ifdef identifier new-line group_opt
  # ifndef identifier new-line group_opt

§ A.14
elif-groups:
  elif-group
  elif-groups elif-group

elif-group:
  # elif constant-expression new-line group_opt

else-group:
  # else new-line group_opt

endif-line:
  # endif new-line

text-line:
  pp-tokens_opt new-line

conditionally-supported-directive:
  pp-tokens new-line

lparen:
  a ( character not immediately preceded by white-space

identifier-list:
  identifier
  identifier-list , identifier

replacement-list:
  pp-tokens_opt

pp-tokens:
  preprocessing-token
  pp-tokens preprocessing-token

new-line:
  the new-line character

defined-macro-expression:
  defined identifier
  defined ( identifier )

h-preprocessing-token:
  any preprocessing-token other than >

h-pp-tokens:
  h-preprocessing-token
  h-pp-tokens h-preprocessing-token

has-include-expression:
  __has_include ( < h-char-sequence > )
  __has_include ( " q-char-sequence " )
  __has_include ( string-literal )
  __has_include ( < h-pp-tokens > )
Annex B  (informative)
Implementation quantities  [implimits]

1 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.

2 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 (9.3), iteration control structures (9.5), and selection control structures (9.4) [256].

(2.2) — Nesting levels of conditional inclusion (19.1) [256].

(2.3) — Pointer (11.3.1), array (11.3.4), and function (11.3.5) declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].

(2.4) — Nesting levels of parenthesized expressions (8.4.3) within a full-expression [256].

(2.5) — Number of characters in an internal identifier (5.10) or macro name (19.3) [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 (11.5) introduced in one declaration [256].

(2.10) — Macro identifiers (19.3) simultaneously defined in one translation unit [65536].

(2.11) — Parameters in one function definition (11.4.1) [256].

(2.12) — Arguments in one function call (8.5.1.2) [256].

(2.13) — Parameters in one macro definition (19.3) [256].

(2.14) — Arguments in one macro invocation (19.3) [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 (19.2) [256].

(2.19) — Case labels for a switch statement (9.4.2) (excluding those for any nested switch statements) [16384].

(2.20) — Data members in a single class (12.2) [16384].

(2.21) — Lambda-captures in one lambda-expression (8.4.5.2) [256].

(2.22) — Enumeration constants in a single enumeration (10.2) [4096].

(2.23) — Levels of nested class definitions (12.2.5) in a single member-specification [256].

(2.24) — Functions registered by atexit() (21.5) [32].

(2.25) — Functions registered by at_quick_exit() (21.5) [32].

(2.26) — Direct and indirect base classes (Clause 13) [16384].

(2.27) — Direct base classes for a single class (Clause 13) [1024].

(2.28) — Members declared in a single class (12.2) [4096].

(2.29) — Final overriding virtual functions in a class, accessible or not (13.3) [16384].

(2.30) — Direct and indirect virtual bases of a class (13.1) [1024].
— Static members of a class (12.2.3) [1024].
— Friend declarations in a class (14.3) [4096].
— Access control declarations in a class (14.1) [4096].
— Member initializers in a constructor definition (15.6.2) [6144].
— `initializer-clauses` in one `braced-init-list` (11.6) [16384].
— Scope qualifications of one identifier (8.4.4.2) [256].
— Nested external specifications [1024].
— Recursive constexpr function invocations (10.1.5) [512].
— Full-expressions evaluated within a core constant expression (8.6) [1048576].
— Template arguments in a template declaration (17.1) [1024].
— Recursively nested template instantiations (17.8.1), including substitution during template argument deduction (17.9.2) [1024].
— Handlers per try block (18.3) [256].
— Number of placeholders (23.14.11.4) [10].
Annex C  (informative)
Compatibility

C.1  C++ and ISO C

1 This subclause lists the differences between C++ and ISO C, by the chapters of this document.

C.1.1 Clause 5: lexical conventions

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')` 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 char16_t
string literal is changed from “array of some-integer-type” to “array of const char16_t”. The type of a
char32_t 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";  // OK: cast added
    f(p);
    f((char*)"def");         // OK: cast added
}
```

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: basic concepts [diff.basic]

1 Affected subclause: 6.1
Change: C++ does not have “tentative definitions” as in C.
E.g., at file scope,

```
int i;
int j;
```

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,

```
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 10.1.7]
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: standard conversions [diff.conv]

1 Affected subclause: 7.11
Change: Converting void* to a pointer-to-object type requires casting.
```
char a[10];
```
ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.

**Rationale:** C++ tries harder than C to enforce compile-time type safety.

**Effect on original feature:** Deletion of semantically well-defined feature.

**Difficulty of converting:** Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast. For example:

```c
char* c = (char*) b;
```

**How widely used:** This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

### C.1.4 Clause 8: expressions

#### 1 Affected subclause: 8.5.1.2

**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.

#### 2 Affected subclause: 8.5.1.6, 8.5.2.2

**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.

#### 3 Affected subclause: 8.5.2.3, 8.5.3

**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,

```c
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.

#### 4 Affected subclause: 8.5.16, 8.5.18, 8.5.19

**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, functions may return lvalues.

**Effect on original feature:** Change to semantics of well-defined feature. Some C expressions that implicitly rely on lvalue-to-rvalue conversions will yield different results. For example,

```c
char arr[100];
sizeof(0, arr)
```

yields 100 in C++ and `sizeof(char*)` in C.

**Difficulty of converting:** Programs must add explicit casts to the appropriate rvalue.

**How widely used:** Rare.

### C.1.5 Clause 9: statements

#### 1 Affected subclause: 9.4.2, 9.6.4

**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.

Affected subclause: 9.6.3

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.1.6 Clause 10: declarations

Affected subclause: 10.1.1

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 {
    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.

Affected subclause: 10.1.1

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.

Affected subclause: 10.1.3

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 namel { /* ... */ } namel; // 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:
class name { /* ... */ }
name i; // i has type class name

Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.
How widely used: Seldom.

4 Affected subclause: 10.1.7 [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: 10.1.7
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++ :

```
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: 10.1.7.4
Change: The keyword auto cannot be used as a storage class specifier.

```
void f() {
   auto int x; // valid C, invalid C++
}
```

Rationale: Allowing the use of auto to deduce the type of a variable from its initializer results in undesired interpretations of auto as a storage class specifier in certain contexts.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Rare.

7 Affected subclause: 10.2
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:

```
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.

8 Affected subclause: 10.2
Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is int.
Example:
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.1.7 Clause 11: declarators

Affected subclause: 11.3.5
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: 11.3.5 [see 8.5.2.3]
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: 11.4
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: 11.6.1
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:
struct A { int x, y; }
struct B { struct A a; }
struct A a = {.y = 1, .x = 2};  // valid C, invalid C++
int arr[3] = {{1}} = 5); // valid C, invalid C++
struct B b = {.a.x = 0}; // valid C, invalid C++
struct A c = {.x = 1, 2}; // valid C, invalid C++

Rationale: In C++, members are destroyed in reverse construction order and the elements of an initializer list are evaluated in lexical order, so field initializers must be specified in order. Array designators conflict with lambda-expression syntax. Nested designators are seldom used.

Effect on original feature: Deletion of feature that is incompatible with C++.

Difficulty of converting: Syntactic transformation.

How widely used: Out-of-order initializers are common. The other features are seldom used.

Affected subclause: 11.6.2
Change: In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating ‘\0’) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string-terminating ‘\0’.

Example:
```c
char array[4] = "abcd";  // valid C, invalid C++
```

Rationale: When these non-terminated arrays are manipulated by standard string functions, there is potential for major catastrophe.

Effect on original feature: Deletion of semantically well-defined feature.

Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating ‘\0’.

How widely used: Seldom. This style of array initialization is seen as poor coding style.

C.1.8 Clause 12: classes

Affected subclause: 12.1 [see also 10.1.3]
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.

Affected subclause: 12.2.4
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.

3 **Affected subclause:** 12.2.5

**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.

4 **Affected subclause:** 12.2.6

**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.1.9 **Clause 15: special member functions**

1 **Affected subclause:** 15.8

**Change:** Copying volatile objects.

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```c
struct X { int i; };
volatile struct X x1 = {0};
struct X x2 = x1; // invalid C++
struct X x3;
```
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.

C.1.10 Clause 19: preprocessing directives

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

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

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, as the following example illustrates.

```
#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.

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.

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: standard conversions

1. **Affected subclause:** 7.11
2. **Change:** Only literals are integer null pointer constants.
3. **Rationale:** Removing surprising interactions with templates and constant expressions.
4. **Effect on original feature:** Valid C++ 2003 code may fail to compile or produce different results in this International Standard, as the following example illustrates:

```cpp
void f(void *); // #1
void f(...); // #2
template<int N> void g() {
    f(0*N);
    // calls #2; used to call #1
}
```

### C.2.3 Clause 8: expressions

1. **Affected subclause:** 8.5.5
2. **Change:** Specify rounding for results of integer `/` and `%`.
3. **Rationale:** Increase portability, C99 compatibility.
4. **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.

### C.2.4 Clause 10: declarations

1. **Affected subclause:** 10.1
2. **Change:** Remove `auto` as a storage class specifier.
3. **Rationale:** New feature.
4. **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.

### C.2.5 Clause 11: declarators

1. **Affected subclause:** 11.6.4
2. **Change:** Narrowing restrictions in aggregate initializers.
3. **Rationale:** Catches bugs.
4. **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 };  // previously valid, now ill-formed
```

### C.2.6 Clause 15: special member functions

1. **Affected subclause:** 15.1, 15.4, 15.8
2. **Change:** Implicitly-declared special member functions are defined as deleted when the implicit definition would have been ill-formed.
3. **Rationale:** Improves template argument deduction failure.
4. **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:** 15.4
3. **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.2.7 Clause 17: templates

1. **Affected subclause:** 17.1
   - **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:** 17.3
   - **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.

   ```
   template <class T> struct X { };
   template <int N> struct Y { };
   X< Y< 1 >> 2 > x;
   ```

3. **Affected subclause:** 17.7.4.2
   - **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.2.8 Clause 20: library introduction

1. **Affected:** Clause 20 – Clause 33
   - **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:** 20.5.1.2
   - **Change:** New headers.
   - **Rationale:** New functionality.
   - **Effect on original feature:** Valid C++ 2003 code that has been compiled expecting `swap` to be in `<algorithm>` may have to instead include `<utility>`.

3. **Affected subclause:** 20.5.3.2
   - **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:** 20.5.4.2.2
   - **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: 20.5.5.3
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.2.9 Clause 21: language support library [diff.cpp03.language.support]
1 Affected subclause: 21.6.2.1
Change: Linking `new` and `delete` operators.
Rationale: The two throwing single-object signatures of `operator new` and `operator delete` are now specified to form the base functionality for the other operators. This clarifies that replacing just these two signatures changes others, even if they are not explicitly changed.
Effect on original feature: Valid C++ 2003 code that replaces global `new` or `delete` operators may execute differently in this International Standard. For example, the following program should write "custom deallocation" twice, once for the single-object delete and once for the array delete.

```cpp
#include <cstdio>
#include <cstdlib>
#include <new>

void* operator new(std::size_t size) throw(std::bad_alloc) {
    return std::malloc(size);
}

void operator delete(void* ptr) throw() {
    std::puts("custom deallocation");
    std::free(ptr);
}

int main() {
    int* i = new int;
    delete i;    // single-object delete
    int* a = new int[3];
    delete [] a;    // array delete
}
```

2 Affected subclause: 21.6.2.1
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.

C.2.10 Clause 22: diagnostics library [diff.cpp03.diagnostics]
1 Affected subclause: 22.4
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.2.11 Clause 23: general utilities library [diff.cpp03.utilities]
1 Affected subclause: 23.10.5
Change: Minimal support for garbage-collected regions.
Rationale: Required by new feature.
Effect on original feature: Valid C++ 2003 code, compiled without traceable pointer support, that interacts with newer C++ code using regions declared reachable may have different runtime behavior.

Change: Standard function object types no longer derived from `std::unary_function` or `std::binary_function`.
Rationale: Superseded by new feature; `unary_function` and `binary_function` are no longer defined.
Effect on original feature: Valid C++ 2003 code that depends on function object types being derived from unary_function or binary_function may fail to compile in this International Standard.

C.2.12 Clause 24: strings library

1. Affected subclause: 24.3
   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.

2. Affected subclause: 24.3.2.1
   Change: Loosen basic_string invalidation rules.
   Rationale: Allow small-string optimization.
   Effect on original feature: Valid C++ 2003 code may execute differently in this International Standard. Some const member functions, such as data and c_str, no longer invalidate iterators.

C.2.13 Clause 26: containers library

1. Affected subclause: 26.2
   Change: Complexity of size() member functions now constant.
   Rationale: Lack of specification of complexity of size() resulted in divergent implementations with inconsistent performance characteristics.
   Effect on original feature: Some container implementations that conform to C++ 2003 may not conform to the specified size() requirements in this International Standard. Adjusting containers such as std::list to the stricter requirements may require incompatible changes.

2. Affected subclause: 26.2
   Change: Requirements change: relaxation.
   Rationale: Clarification.
   Effect on original feature: Valid C++ 2003 code that attempts to meet the specified container requirements may now be over-specified. Code that attempted to be portable across containers may need to be adjusted as follows:

   (2.1) not all containers provide size(); use empty() instead of size() == 0;
   (2.2) not all containers are empty after construction (array);
   (2.3) not all containers have constant complexity for swap() (array).

3. Affected subclause: 26.2
   Change: Requirements change: default constructible.
   Rationale: Clarification of container requirements.
   Effect on original feature: Valid C++ 2003 code that attempts to explicitly instantiate a container using a user-defined type with no default constructor may fail to compile.

4. Affected subclause: 26.2.3, 26.2.6
   Change: Signature changes: from void return types.
   Rationale: Old signature threw away useful information that may be expensive to recalculate.
   Effect on original feature: The following member functions have changed:

   (4.1) erase(iter) for set, multiset, map, multimap
   (4.2) erase(begin, end) for set, multiset, map, multimap
   (4.3) insert(pos, num, val) for vector, deque, list, forward_list
   (4.4) insert(pos, beg, end) for vector, deque, list, forward_list

   Valid C++ 2003 code that relies on these functions returning void (e.g., code that creates a pointer to member function that points to one of these functions) will fail to compile with this International Standard.

5. Affected subclause: 26.2.3, 26.2.6
   Change: Signature changes: from iterator to const_iterator parameters.
   Rationale: Overspecification.
   Effect on original feature: The signatures of the following member functions changed from taking an iterator to taking a const_iterator:

   (5.1) insert(iter, val) for vector, deque, list, set, multiset, map, multimap

§ C.2.13
Valid C++ 2003 code that uses these functions may fail to compile with this International Standard.

6 Affected subclause: 26.2.3, 26.2.6
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.2.14 Clause 28: algorithms library [diff.cpp03.algorithms]

1 Affected subclause: 28.1
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.2.15 Clause 29: numerics library [diff.cpp03.numerics]

1 Affected subclause: 29.5
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.2.16 Clause 30: input/output library [diff.cpp03.input.output]

1 Affected subclause: 30.7.4.1.3, 30.7.5.1.3, 30.5.5.4
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.

2 Affected subclause: 30.5.3.1.1
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.

3 Affected subclause: 30.5.3
Change: Flag types in std::ios_base are now bitmasks with values defined as const expr 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 [diff.cpp11]
1 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 [diff.cpp11.lex]
1 Affected subclause: 5.9
Change: **pp-number** can contain one or more single quotes.
Rationale: Necessary to enable single quotes as digit separators.
Effect on original feature: Valid C++ 2011 code may fail to compile or may change meaning in this International Standard. For example, the following code is valid both in C++ 2011 and in this International Standard, but the macro invocation produces different outcomes because the single quotes delimit a character literal in C++ 2011, whereas they are digit separators in this International Standard:

```c
#define M(x, ...) __VA_ARGS__
int x[2] = { M(1'2,3'4, 5) };
// int x[2] = { 3'4, 5 }; — this International Standard
```

C.3.2 Clause 6: basic concepts [diff.cpp11.basic]
1 Affected subclause: 6.6.4.4.2
Change: New usual (non-placement) deallocator.
Rationale: Required for sized deallocation.
Effect on original feature: Valid C++ 2011 code could declare a global placement allocation function and deallocation function as follows:

```c
void* operator new(std::size_t, std::size_t);
void operator delete(void*, std::size_t) noexcept;
```
In this International Standard, however, the declaration of `operator delete` might match a predefined usual (non-placement) `operator delete` (6.6.4.4). If so, the program is ill-formed, as it was for class member allocation functions and deallocation functions (8.5.2.4).

C.3.3 Clause 8: expressions [diff.cpp11.expr]
1 Affected subclause: 8.5.16
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 (7.1), array-to-pointer (7.2), and function-to-pointer (7.3) standard conversions), especially the creation of the temporary due to lvalue-to-rvalue conversion, were considered gratuitous and surprising.
Effect on original feature: Valid C++ 2011 code that relies on the conversions may behave differently in this International Standard:

```c
struct S {
    int x = 1;
    void mf() { x = 2; }
};
int f(bool cond) {
    S s;
    (cond ? s : throw 0).mf();
    return s.x;
}
```
In C++ 2011, `f(true)` returns 1. In this International Standard, it returns 2.

```c
sizeof(true ? "" : throw 0)
```
In C++ 2011, the expression yields `sizeof(const char*)`. In this International Standard, it yields `sizeof(const char[1])`.

§ C.3.3 1295
C.3.4 Clause 10: declarations

Affected subclause: 10.1.5
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:

```cpp
struct S {
    constexpr const int &f();
    int &f();
};
```

C.3.5 Clause 11: declarators

Affected subclause: 11.6.1
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.

```cpp
struct S {
    //Aggregate in C++ 2014 onwards.
    int m = 1;
};
struct X {
    operator int();
    operator S();
};
X a{};
S b{a}; // uses copy constructor in C++ 2011,
//performs aggregate initialization in this International Standard
```

C.3.6 Clause 20: library introduction

Affected subclause: 20.5.1.2
Change: New header.
Rationale: New functionality.
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.3.7 Clause 30: input/output library

Affected subclause: 30.12
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

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

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.

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.

```c++
#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 8: expressions

1. Affected subclause: 8.5.1.6, 8.5.2.2
   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 subclause: 8.5.2.4, 8.5.2.5
   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 10: declarations

1. Affected subclause: 10.1.1
   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: 10.1.7.4
   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:
   ```c++
   auto x1{};  // was std::initializer_list<int>, now int
   auto x2{1, 2};  // was std::initializer_list<int>, now ill-formed
   ```

C.4.4 Clause 11: declarators

1. Affected subclause: 11.3.5
   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:
   ```c++
   void g1() noexcept;
   void g2();
   template<class T> int f(T *, T *);
   int x = f(g1, g2);  // ill-formed; previously well-formed
   ```

2. Affected subclause: 11.6.1
   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:
   ```c++
   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.4.5 Clause 15: special member functions

Affected subclause: 15.6.3
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.

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.4.6 Clause 17: templates

Affected subclause: 17.9.2.5
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:

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.4.7 Clause 18: exception handling

Affected subclause: 18.4
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 terminate rather than unexpected and might not perform stack unwinding prior to such a call.

C.4.8 Clause 20: library introduction

Affected subclause: 20.5.1.2
Change: New headers.
Rationale: New functionality.
Effect on original feature: The following C++ headers are new: <any>, <execution>, <filesystem>, <memory_resource>, <optional>, <string_view>, and <variant>. Valid C++ 2014 code that #includes headers with these names may be invalid in this International Standard.
Affected subclause: 20.5.4.2.3
Change: New reserved namespaces.
Rationale: Reserve namespaces for future revisions of the standard library that might otherwise be incompatible with existing programs.
Effect on original feature: The global namespaces `std` followed by an arbitrary sequence of digits is reserved for future standardization. Valid C++ 2014 code that uses such a top-level namespace, e.g., `std2`, may be invalid in this International Standard.

C.4.9 Clause 23: general utilities library

Affected subclause: 23.14.13
Change: Constructors taking allocators removed.
Rationale: No implementation consensus.
Effect on original feature: Valid C++ 2014 code may fail to compile or may change meaning in this International Standard. Specifically, constructing a `std::function` with an allocator is ill-formed and uses-allocator construction will not pass an allocator to `std::function` constructors in this International Standard.

Affected subclause: 23.11.3
Change: Different constraint on conversions from `unique_ptr`.
Rationale: Adding array support to `shared_ptr`, via the syntax `shared_ptr<T[]>` and `shared_ptr<T[N]>`.
Effect on original feature: Valid C++ 2014 code may fail to compile or may change meaning in this International Standard. For example:

```cpp
#include <memory>
std::unique_ptr<int[]> arr(new int[1]);
std::shared_ptr<int> ptr(std::move(arr)); // error: int(*)[] is not compatible with int*
```

C.4.10 Clause 24: strings library

Affected subclause: 24.3.2
Change: Non-const `.data()` member added.
Rationale: The lack of a non-const `.data()` differed from the similar member of `std::vector`. This change regularizes behavior for this International Standard.
Effect on original feature: Overloaded functions which have differing code paths for `char*` and `const char*` arguments will execute differently when called with a non-const string’s `.data()` member in this International Standard.

```cpp
int f(char *) = delete;
int f(const char *);
string s;
int x = f(s.data()); // ill-formed; previously well-formed
```

C.4.11 Clause 26: containers library

Affected subclause: 26.2.6
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.4.12 Annex D: compatibility features

Change: The class templates `auto_ptr`, `unary_function`, and `binary_function`, the function templates `random_shuffle`, and the function templates (and their return types) `ptr_fun`, `mem_fun`, `mem_fun_ref`, `bind1st`, and `bind2nd` are not defined.

Rationale: Superseded by new features.

Effect on original feature: Valid C++ 2014 code that uses these class templates and function templates may fail to compile in this International Standard.

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

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

Affected subclause: 5.11
Change: New keywords.

Rationale: Required for new features. The `requires` keyword is added to introduce constraints through a `requires-clause` or a `requires-expression`. The `concept` keyword is added to enable the definition of concepts (17.6.8).

Effect on original feature: Valid ISO C++ 2017 code using `concept` or `requires` as an identifier is not valid in this International Standard.

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 {}; // ill-formed; previously well-formed
  Y<operator<=> y;
}
```

C.5.2 Clause 8: expressions

Affected subclause: 8.4.5.2
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.5.3 Clause 17: templates

Affected subclause: 17.2
Change: An `unqualified-id` that is followed by a `<` and for which name lookup finds nothing or finds a function will be treated as a `template-name` in order to potentially cause argument dependent lookup to be performed.

Rationale: It was problematic to call a function template with an explicit template argument list via argument dependent lookup because of the need to have a template with the same name visible via normal lookup.

Effect on original feature: Previously valid code that uses a function name as the left operand of a `<` operator would become ill-formed.

```cpp
struct A {};
bool operator<(void (*)(void), A);
```
void f() {}
int main() {
A a;
f < a; // ill-formed; previously well-formed
(f) < a; // still well formed
}

C.5.4 Clause 20: library introduction
[diff.cpp17.library]
20.5.1.2
Change: New headers.
Rationale: New functionality.
Effect on original feature: The following C++ headers are new: <compare> and <syncstream>. Valid
C++ 2017 code that #includes headers with these names may be invalid in this International Standard.

C.6 C standard library
[diff.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
[diff.mods.to.headers]
1 For compatibility with the C standard library, the C++ standard library provides the C headers enumerated
in D.5, but their use is deprecated in C++.
2 There are no C++ headers for the C headers <stdatomic.h>, <stdnoreturn.h>, and <threads.h>, nor are the
C headers themselves part of C++.
3 The C++ headers <complex> (D.4.1) and <ctgmath> (D.4.4), as well as their corresponding 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.
4 The headers <ciso646>, <cstdaqian>, and <stdbool> (D.4.3) are meaningless in C++. Use of the
C++ headers <complex>, <cstdaqian>, and <stdbool> is deprecated (D.5).

C.6.2 Modifications to definitions
[diff.mods.to.definitions]
C.6.2.1 Types char16_t and char32_t
[diff.char16]
1 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> (24.5.5).

C.6.2.2 Type wchar_t
[diff.wchar.t]
1 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
<stddef> (21.2.1), <stdlib> (21.2.2), or <wchar> (24.5.4).

C.6.2.3 Header <cassert.h>
[diff.header.assert.h]
1 The token static_assert is a keyword in this International Standard (5.11). It does not appear as a macro
name defined in <cassert> (22.3.1).

C.6.2.4 Header <iso646.h>
[diff.header.iso646.h]
1 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). They do not appear as macro names defined in <iso646>.

C.6.2.5 Header <cstdlib.h>
[diff.headercstdlib.h]
1 The token alignas is a keyword in this International Standard (5.11). It does not appear as a macro name
defined in <cstdlib> (D.4.2).

C.6.2.6 Header <stdbool.h>
[diff.headerstdbool.h]
1 The tokens bool, true, and false are keywords in this International Standard (5.11). They do not appear
as macro names defined in <stdbool> (D.4.3).
C.6.2.7 Macro NULL

The macro NULL, defined in any of `<locale>` (25.5), `<cstdlib>` (21.2.1), `<cint>` (30.12.1), `<stdio>` (21.2.1), `<string>` (24.5.3), `<ctime>` (23.17.8), or `<wchar>` (24.5.4), is an implementation-defined C++ null pointer constant in this International Standard (21.2).

C.6.3 Modifications to declarations

Header `<cstring>` (24.5.3): The following functions have different declarations:

- `strchr`
- `strpbrk`
- `strrchr`
- `strstr`
- `memchr`

Subclause 24.5.3 describes the changes.

Header `<cwchar>` (24.5.4): The following functions have different declarations:

- `wcschr`
- `wcspbrk`
- `wcsrchr`
- `wcsstr`
- `wmemchr`

Subclause 24.5.4 describes the changes.

Header `<cstdlib>` (21.2.2) 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>` (21.2.2): The following functions have different behavior:

- `atexit`
- `exit`
- `abort`

Subclause 21.5 describes the changes.

Header `<csetjmp>` (21.11.2): The following functions have different behavior:

- `longjmp`

Subclause 21.11.2 describes the changes.

C.6.4.1 Macro offsetof(type, member-designator)

The macro `offsetof`, defined in `<stddef>` (21.2.1), accepts a restricted set of `type` arguments in this International Standard. Subclause 21.2.4 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 23.10.12 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 (10.6.4).

D.1 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 redeclared 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.2 Implicit declaration of copy functions  [depr.impldec]
1 The implicit definition of a copy constructor as defaulted is deprecated if the class has a user-declared copy assignment operator or a user-declared destructor. The implicit definition of a copy assignment operator as defaulted is deprecated if the class has a user-declared copy constructor or a user-declared destructor (15.4, 15.8). In a future revision of this International Standard, these implicit definitions could become deleted (11.4).

D.3 Deprecated exception specifications  [depr.except.spec]
1 The noexcept-specifier throw() is deprecated.

D.4 C++ standard library headers  [depr.cpp.headers]
1 For compatibility with prior C++ International Standards, the C++ standard library provides headers <ccomplex> (D.4.1), <cstdalign> (D.4.2), <cstdbool> (D.4.3), and <ctgmath> (D.4.4). The use of these headers is deprecated.

D.4.1 Header <ccomplex> synopsis  [depr.ccomplex.syn]
1 The header <ccomplex> behaves as if it simply includes the header <complex> (29.5.1).

D.4.2 Header <cstdalign> synopsis  [depr.cstdalign.syn]
1 The contents of the header <cstdalign> are the same as the C standard library header <stdalign.h>, with the following changes: The header <cstdalign> and the header <stdalign.h> shall not define a macro named alignas.

See also: ISO C 7.15

D.4.3 Header <cstdbool> synopsis  [depr.cstdbool.syn]
1 The contents of the header <cstdbool> are the same as the C standard library header <stdbool.h>, with the following changes: The header <cstdbool> and the header <stdbool.h> shall not define macros named bool, true, or false.

See also: ISO C 7.18
D.4.4 Header <ctgmath> synopsis

```cpp
#include <complex>
#include <cmath>
```

1 The header <ctgmath> simply includes the headers <complex> (29.5.1) and <cmath> (29.9.1).

2 [Note: The overloads provided in C by type-generic macros are already provided in <complex> and <cmath> by “sufficient” additional overloads. — end note]

D.5 C standard library headers

For compatibility with the C standard library, the C++ standard library provides the C headers shown in Table 141.

| <assert.h> | <inttypes.h> | <signal.h> | <stdio.h> | <wchar.h> |
| <complex.h> | <iso646.h> | <stdalign.h> | <stdlib.h> | <wctype.h> |
| <type.h> | <limits.h> | <stdarg.h> | <string.h> | <ctype.h> |
| <errno.h> | <locale.h> | <stdbool.h> | <tgmath.h> | <fenv.h> |
| <float.h> | <setjmp.h> | <stdbool.h> | <uchar.h> | <math.h> |

The header <complex.h> behaves as if it simply includes the header <ccomplex>. The header <tgmath.h> behaves as if it simply includes the header <ctgmath>.

3 Every other C header, each of which has a name of the form name.h, behaves as if each name placed in the standard library namespace by the corresponding c name header is placed within the global namespace scope, except for the functions described in 29.9.5, the declaration of std::byte (21.2.1), and the functions and function templates described in 21.2.5. 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 (10.3.3).

[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.6 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&);
}
```

2 To avoid redundant definitions of operator!= out of operator== and operators >, <=, and >= out of operator<, the library provides the following:

```cpp
template<class T> bool operator!=(const T x, const T y);
```

3 Requires: Type T is EqualityComparable (Table 20).

4 Returns: !(x == y).

```cpp
template<class T> bool operator> (const T x, const T y);
```

5 Requires: Type T is LessThanComparable (Table 21).

6 Returns: y < x.

```cpp
template<class T> bool operator< (const T x, const T y);
```

7 Requires: Type T is LessThanComparable (Table 21).

8 Returns: !(y < x).
template<class T> bool operator>=(const T& x, const T& y);

Requires: Type T is LessThanComparable (Table 21).
Returns: !(x < y).

D.7 char* streams

The header <strstream> defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.7.1 Class strstreambuf
	namespace std {
            class strstreambuf : public basic_streambuf<char> {
            public:
                explicit strstreambuf(streamsize alsize_arg = 0);
                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(const char* gnext_arg, streamsize n);
                strstreambuf(const signed char* gnext_arg, streamsize n);
                strstreambuf(const unsigned char* gnext_arg, streamsize n);
                strstreambuf(const unsigned char* gnext_arg, streamsize n);
                virtual ~strstreambuf();
                void freeze(bool freezefl = true);
                char* str();
                int pcount();
            protected:
                int_type overflow (int_type c = EOF) override;
                int_type pbackfail(int_type c = EOF) override;
                int_type underflow() override;
                pos_type seekoff(off_type off, ios_base::seekdir way,
                                 ios_base::openmode which = ios_base::in | ios_base::out) override;
                pos_type seekpos(pos_type sp, ios_base::openmode which = ios_base::in | ios_base::out) override;
                streambuf* setbuf(char* s, streamsize n) override;
            private:
                using strstate = T1; // exposition only
                static const strstate allocated; // exposition only
                static const strstate constant; // exposition only
                static const strstate dynamic; // exposition only
                static const strstate frozen; // exposition only
                strstate strmstate; // 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;
constant, set when the array object has \texttt{const} elements, so the output sequence cannot be written;

dynamic, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

frozen, set when the program has requested that the array object not be altered, reallocated, or freed.

[Note: For the sake of exposition, the maintained data is presented here as:

strstate strmode, the attributes of the array object associated with the \texttt{strstreambuf} object;

int alsizet, the suggested minimum size for a dynamic array object;

void* (*palloc)(size_t), points to the function to call to allocate a dynamic array object;

void (*pfree)(void*), points to the function to call to free a dynamic array object.

—end note]

Each object of class \texttt{strstreambuf} has a seekable area, delimited by the pointers \texttt{seeklow} and \texttt{seekhigh}. If \texttt{gnext} is a null pointer, the seekable area is undefined. Otherwise, \texttt{seeklow} equals \texttt{gbeg} and \texttt{seekhigh} is either \texttt{pend}, if \texttt{pend} is not a null pointer, or \texttt{gend}.

D.7.1.1 \texttt{strstreambuf} constructors

```c
explicit strstreambuf(streamsize alsize_arg = 0);
```

Effects: Constructs an object of class \texttt{strstreambuf}, initializing the base class with \texttt{streambuf()}. The postconditions of this function are indicated in Table 142.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsizet</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>

```c
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

Effects: Constructs an object of class \texttt{strstreambuf}, initializing the base class with \texttt{streambuf()}. The postconditions of this function are indicated in Table 143.

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>dynamic</td>
</tr>
<tr>
<td>alsizet</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>

```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 \texttt{strstreambuf}, initializing the base class with \texttt{streambuf()}. The postconditions of this function are indicated in Table 144.

\texttt{gnext_arg} shall point to the first element of an array object whose number of elements \texttt{N} is determined as follows:

- If \texttt{n > 0}, \texttt{N} is \texttt{n}.
- If \texttt{n == 0}, \texttt{N} is \texttt{std::strlen(gnext_arg)}. 

§ D.7.1.1
Table 144 — strstreambuf(charT*, streamsize, charT*) effects

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>strmode</td>
<td>0</td>
</tr>
<tr>
<td>alsizze</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>

(4.3) If \( n < 0 \), \( N \) is INT_MAX.334

5 If pbeg_arg is a null pointer, the function executes:
   setg(gnext_arg, gnext_arg, gnext_arg + N);

6 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);

7 Effects: Behaves the same as strstreambuf((char*)gnext_arg,n), except that the constructor also sets constant in strmode.

virtual ~strstreambuf();

8 Effects: Destroys an object of class strstreambuf. The function frees the dynamically allocated array object only if \((\text{strmode} & \text{allocated}) != 0 \) and \((\text{strmode} & \text{frozen}) == 0 \). (D.7.1.3 describes how a dynamically allocated array object is freed.)

D.7.1.2 Member functions

void freeze(bool freezefl = true);

1 Effects: If strmode & dynamic is nonzero, alters the freeze status of the dynamic array object as follows:

(1.1) If freezefl is true, the function sets frozen in strmode.

(1.2) Otherwise, it clears frozen in strmode.

char* str();

2 Effects: Calls freeze(), then returns the beginning pointer for the input sequence, gbeg.

3 Remarks: The return value can be a null pointer.

int pcount() const;

4 Effects: If the next pointer for the output sequence, pnext, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, pnext - pbeg.

D.7.1.3 strstreambuf overridden virtual functions

int_type overflow(int_type c = EOF) override;

1 Effects: Appends the character designated by c to the output sequence, if possible, in one of two ways:

(1.1) If c != EOF and if either the output sequence has a write position available or the function makes a write position available (as described below), assigns c to *pnext++. Returns (unsigned char)c.

(1.2) If c == EOF, there is no character to append. Returns a value other than EOF.

334) The function signature strlen(const char*) is declared in <cstring> (24.5.3). The macro INT_MAX is defined in <climits> (21.3.5).
Returns EOF to indicate failure.

Remarks: The function can alter the number of write positions available as a result of any call.

To make a write position available, the function reallocates (or initially allocates) an array object with
a sufficient number of elements \( n \) to hold the current array object (if any), plus at least one additional
write position. How many additional write positions are made available is otherwise unspecified. \(^{335}\) If
\texttt{palloc} is not a null pointer, the function calls \( (*\texttt{palloc})(n) \) to allocate the new dynamic array object.
Otherwise, it evaluates the expression \texttt{new charT[n]}. In either case, if the allocation fails, the function
returns EOF. Otherwise, it sets \texttt{allocated\texttt{strmode}}.

To free a previously existing dynamic array object whose first element address is \texttt{p}: If \texttt{pfree} is not a
null pointer, the function calls \( (*\texttt{pfree})(p) \). Otherwise, it evaluates the expression \texttt{delete[]p}.

If \( (\texttt{strmode} \& \texttt{dynamic}) == 0 \), or if \( (\texttt{strmode} \& \texttt{frozen}) \neq 0 \), the function cannot extend the array
(reallocate it with greater length) to make a write position available.

\texttt{int_type pbackfail(int_type c = EOF) override;}

Puts back the character designated by \texttt{c} to the input sequence, if possible, in one of three ways:

- If \texttt{c} \neq EOF, if the input sequence has a putback position available, and if \((\texttt{char})c == \texttt{gnext[-1]}\),
  assigns \( \texttt{gnext} - 1 \) to \texttt{gnext}.
  Returns \texttt{c}.

- If \texttt{c} \neq EOF, if the input sequence has a putback position available, and if \texttt{strmode \& constant}
is zero, assigns \texttt{c} to \( *--\texttt{gnext} \).
  Returns \texttt{c}.

- If \texttt{c} \neq EOF and if the input sequence has a putback position available, assigns \texttt{gnext} - 1 to
  \texttt{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.

\texttt{int_type underflow() override;}

Effects: Reads a character from the input sequence, if possible, without moving the stream position
past it, as follows:

- If the input sequence has a read position available, the function signals success by returning
  \((\texttt{unsigned char})*\texttt{gnext}\).

- Otherwise, if the current read next pointer \( \texttt{pnext} \) is not a null pointer and is greater than the
  current read end pointer \( \texttt{gend} \), makes a read position available by assigning to \( \texttt{gend} \) a value greater
  than \( \texttt{gnext} \) and no greater than \( \texttt{pnext} \).
  Returns \((\texttt{unsigned char})*\texttt{gnext}\).

Returns EOF to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

\texttt{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 145.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails.
Otherwise, the function determines \texttt{newoff} as indicated in Table 146.

If \((\texttt{newoff} + \texttt{off}) \leq (\texttt{seeklow} - \texttt{xbeg}) \) or \((\texttt{seekhigh} - \texttt{xbeg}) \leq (\texttt{newoff} + \texttt{off}) \), the positioning
operation fails. Otherwise, the function assigns \texttt{xbeg} + \texttt{newoff} + \texttt{off} to the next pointer \( \texttt{xnext} \).

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))}.

\(^{335}\) An implementation should consider \texttt{alsize} in making this decision.
pos_type seekpos(pos_type sp, ios_base::openmode which
= ios_base::in | ios_base::out) override;

Effects: Alters the stream position within one of the controlled sequences, if possible, to correspond to
the stream position stored in sp (as described below).

(17.1) — If (which & ios::in) != 0, positions the input sequence.
(17.2) — If (which & ios::out) != 0, positions the output sequence.
(17.3) — If the function positions neither sequence, the positioning operation fails.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails.
Otherwise, the function determines newoff from sp.offset():

(18.1) — If newoff is an invalid stream position, has a negative value, or has a value greater than (seekhigh
- seeklow), the positioning operation fails
(18.2) — Otherwise, the function adds newoff to the beginning pointer xbeg and stores the result in the
next pointer xnext.

Returns: pos_type(newoff), constructed from the resultant offset newoff (of type off_type), that
stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed
object cannot represent the resultant stream position, the return value is pos_type(off_type(-1)).

streambuf<char>* setbuf(char* s, streamsize n) override;

Effects: Implementation defined, except that setbuf(0, 0) has no effect.

D.7.2 Class istrstream

namespace std {
    class istrstream : public basic_istream<char> {
        public:
            explicit istrstream(const char* s);
            explicit istrstream(char* s);
            istrstream(const char* s, streamsize n);
            istrstream(char* s, streamsize n);
            virtual ~istrstream();

            strstreambuf* rdbuf() const;
            char* str();

§ D.7.2 1309
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:

\[
\begin{align*}
&\text{— } \text{sb, the } \text{strstreambuf} \text{ object.}
\end{align*}
\]

D.7.2.1 `istrstream` constructors

```cpp
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`.

```cpp
istrstream(const char* s, streamsize n);
istrstream(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.7.2.2 Member functions

```cpp
strstreambuf* rdbuf() const;
char* str();
```

**Returns:**
- `const_cast<strstreambuf*>(&sb)`
- `rdbuf()->str()`

D.7.3 Class `ostrstream`

```cpp
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:

\[
\begin{align*}
&\text{— } \text{sb, the } \text{strstreambuf} \text{ object.}
\end{align*}
\]

D.7.3.1 `ostrstream` constructors

```cpp
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.7.3.2 Member functions
strstreambuf* rdbuf() const;

Returns: (strstreambuf*)&sb.

void freeze(bool freezefl = true);

Effects: Calls rdbuf()->freeze(freezefl).

char* str();

Returns: rdbuf()->str().

int pcount() const;

Returns: rdbuf()->pcount().

D.7.4 Class strstream
namespace std {
    class strstream : public basic_iostream<char> {
    public:
    // types
    using char_type = char;
    using int_type = char_traits<char>::int_type;
    using pos_type = char_traits<char>::pos_type;
    using off_type = char_traits<char>::off_type;
    // constructors/destructor
    strstream();
    strstream(char* s, int n,
    ios_base::openmode mode = ios_base::in|ios_base::out);
    virtual ~strstream();
    // members
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    int pcount() const;
    char* str();

    private:
    strstreambuf sb; // exposition only
    }
};

The class strstream supports reading and writing from objects of class strstreambuf. It supplies a strstreambuf object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

sb, the strstreambuf object.

336) The function signature strlen(const char*) is declared in <cstring> (24.5.3).
D.7.4.1 *strstream constructors*  

```c++
strstream();
```

*Effects:* Constructs an object of class *strstream*, initializing the base class with *iostream(&sb)*.

```c++
strstream(char* s, int n,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

*Effects:* Constructs an object of class *strstream*, initializing the base class with *iostream(&sb)* and initializing *sb* with one of the two constructors:

1. If *(mode & app) == 0*, then *s* shall designate the first element of an array of *n* elements. The constructor is *strstreambuf(s,n,s)*.
2. If *(mode & app) != 0*, then *s* shall designate the first element of an array of *n* elements that contains an NIBS whose first element is designated by *s*. The constructor is *strstreambuf(s,n,s + std::strlen(s))*. 

D.7.4.2 *strstream destructor*  

```c++
virtual ~strstream();
```

*Effects:* Destroys an object of class *strstream*.

D.7.4.3 *strstream operations*  

```c++
strstreambuf* rdbuf() const;
```

*Returns:* &*sb*.

```c++
void freeze(bool freezeflag = true);
```

*Effects:* Calls *rdbuf()*--freeze(*freezeflag*).

```c++
char* str();
```

*Returns:* *rdbuf()*--str()

```c++
int pcount() const;
```

*Returns:* *rdbuf()*--pcount()

D.8 *uncaught_exception*  

```c++
namespace std {
    bool uncaught_exception() noexcept;
}
```

*Returns:* uncaught_exceptions() > 0.

D.9 Old adaptable function bindings  

D.9.1 Weak result types  

A call wrapper (23.14.2) may have a *weak result type*. If it does, the type of its member type *result_type* is based on the type *T* of the wrapper’s target object:

1. If *T* is a pointer to function type, *result_type* shall be a synonym for the return type of *T*;
2. If *T* is a pointer to member function, *result_type* shall be a synonym for the return type of *T*;
3. If *T* is a class type and the *qualified-id* *T::*result_type is valid and denotes a type (17.9.2), then *result_type* shall be a synonym for *T::*result_type;
4. Otherwise *result_type* shall not be defined.
D.9.2 Typedefs to support function binders

1 To enable old function adaptors to manipulate function objects that take one or two arguments, many of
the function objects in this document correspondingly provide typedef-names argument_type and result_type
for function objects that take one argument and first_argument_type, second_argument_type, and
result_type for function objects that take two arguments.

2 The following member names are defined in addition to names specified in 23.14:

```cpp
namespace std {
    template<class T> struct owner_less<shared_ptr<T>> {
        using result_type = bool;
        using first_argument_type = shared_ptr<T>;
        using second_argument_type = shared_ptr<T>;
    };

    template<class T> struct owner_less<weak_ptr<T>> {
        using result_type = bool;
        using first_argument_type = weak_ptr<T>;
        using second_argument_type = weak_ptr<T>;
    };

    template<class T> class reference_wrapper {
        public:
            using result_type = see below;  // not always defined
            using argument_type = see below;  // not always defined
            using first_argument_type = see below;  // not always defined
            using second_argument_type = see below;  // not always defined
    };

    template<class T> struct plus {
        using first_argument_type = T;
        using second_argument_type = T;
        using result_type = T;
    };

    template<class T> struct minus {
        using first_argument_type = T;
        using second_argument_type = T;
        using result_type = T;
    };

    template<class T> struct multiplies {
        using first_argument_type = T;
        using second_argument_type = T;
        using result_type = T;
    };

    template<class T> struct divides {
        using first_argument_type = T;
        using second_argument_type = T;
        using result_type = T;
    };

    template<class T> struct modulus {
        using first_argument_type = T;
        using second_argument_type = T;
        using result_type = T;
    };

    template<class T> struct negate {
        using argument_type = T;
        using result_type = T;
    };
```

§ D.9.2
template<class T> struct equal_to {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct not_equal_to {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct greater {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct less {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct greater_equal {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct less_equal {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct logical_and {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct logical_or {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = bool;
};

template<class T> struct logical_not {
    using argument_type = T;
    using result_type = bool;
};

template<class T> struct bit_and {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = T;
};

template<class T> struct bit_or {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = T;
};
template<class T> struct bit_xor {
    using first_argument_type = T;
    using second_argument_type = T;
    using result_type = T;
};

template<class T> struct bit_not {
    using argument_type = T;
    using result_type = T;
};

template<class R, class T1> class function<R(T1)> {
public:
    using argument_type = T1;
};

template<class R, class T1, class T2> class function<R(T1, T2)> {
public:
    using first_argument_type = T1;
    using second_argument_type = T2;
};

reference_wrapper<T> has a weak result type (D.9.1). If T is a function type, result_type shall be a synonym for the return type of T.

The template specialization reference_wrapper<T> shall define a nested type named argument_type as a synonym for T1 only if the type T is any of the following:

- a function type or a pointer to function type taking one argument of type T1
- a pointer to member function R T0::f() cv (where cv represents the member function’s cv-qualifiers); the type T1 is cv T0*
- a class type where the qualified-id T::argument_type is valid and denotes a type (17.9.2); the type T1 is T::argument_type.

The template instantiation reference_wrapper<T> shall define two nested types named first_argument_type and second_argument_type as synonyms for T1 and T2, respectively, only if the type T is any of the following:

- a function type or a pointer to function type taking two arguments of types T1 and T2
- a pointer to member function R T0::f(T2) cv (where cv represents the member function’s cv-qualifiers); the type T1 is cv T0*
- a class type where the qualified-ids T::first_argument_type and T::second_argument_type are both valid and both denote types (17.9.2); the type T1 is T::first_argument_type and the type T2 is T::second_argument_type.

All enabled specializations hash<Key> of hash (23.14.15) provide two nested types, result_type and argument_type, which shall be synonyms for size_t and Key, respectively.

The forwarding call wrapper g returned by a call to bind(f, bound_args...) (23.14.11.3) shall have a weak result type (D.9.1).

The forwarding call wrapper g returned by a call to bind<R>(f, bound_args...) (23.14.11.3) shall have a nested type result_type defined as a synonym for R.

The simple call wrapper returned from a call to mem_fn(pm) shall have a nested type result_type that is a synonym for the return type of pm when pm is a pointer to member function.

The simple call wrapper returned from a call to mem_fn(pm) shall define two nested types named argument_type and result_type as synonyms for cv T* and Ret, respectively, when pm is a pointer to member function with cv-qualifier cv and taking no arguments, where Ret is pm’s return type.

The simple call wrapper returned from a call to mem_fn(pm) shall define three nested types named first_argument_type, second_argument_type, and result_type as synonyms for cv T*, T1, and Ret, respectively, when pm is a pointer to member function with cv-qualifier cv and taking one argument of type T1, where Ret is pm’s return type.
The following member names are defined in addition to names specified in Clause 26:

```cpp
namespace std {
    template<class Key, class T, class Compare, class Allocator>
    class map<Key, T, Compare, Allocator>::value_compare {
    public:
        using result_type = bool;
        using first_argument_type = value_type;
        using second_argument_type = value_type;
    };

    template<class Key, class T, class Compare, class Allocator>
    class multimap<Key, T, Compare, Allocator>::value_compare {
    public:
        using result_type = bool;
        using first_argument_type = value_type;
        using second_argument_type = value_type;
    };
}
```

D.9.3 Negators

The header `<functional>` has the following additions:

```cpp
namespace std {
    template<class Predicate> class unary_negate;
    template<class Predicate>
    constexpr unary_negate<Predicate> not1(const Predicate&);
    template<class Predicate> class binary_negate;
    template<class Predicate>
    constexpr binary_negate<Predicate> not2(const Predicate&);
}
```

Negators `not1` and `not2` take a unary and a binary predicate, respectively, and return their logical negations (8.5.2.1).

```cpp
template<class Predicate>
class unary_negate {
public:
    constexpr explicit unary_negate(const Predicate& pred);
    constexpr bool operator()(const typename Predicate::argument_type& x) const;
    using argument_type = typename Predicate::argument_type;
    using result_type = bool;
};
```

Returns: `!pred(x)`.

```cpp
template<class Predicate>
constexpr unary_negate<Predicate> not1(const Predicate& pred);
```

Returns: `unary_negate<Predicate>(pred)`.

```cpp
template<class Predicate>
class binary_negate {
public:
    constexpr explicit binary_negate(const Predicate& pred);
    constexpr bool operator()(const typename Predicate::first_argument_type& x,
                              const typename Predicate::second_argument_type& y) const;
    using first_argument_type = typename Predicate::first_argument_type;
    using second_argument_type = typename Predicate::second_argument_type;
    using result_type = bool;
};
```
constexpr bool operator()(const typename Predicate::first_argument_type& x, const typename Predicate::second_argument_type& y) const;

Returns: !pred(x, y).

template<class Predicate>
constexpr binary_negate<Predicate> not2(const Predicate& pred);

Returns: binary_negate<Predicate>(pred).

D.10 The default allocator

The following members and explicit class template specialization are defined in addition to those specified in 23.10.10:

namespace std {
    // specialization for void
    template<> class allocator<void> {
        public:
            using value_type = void;
            using pointer = void*;
            using const_pointer = const void*;
            // reference-to-void members are impossible.

            template<class U> struct rebind { using other = allocator<U>; };
    };

    template<class T> class allocator {
        public:
            using size_type = size_t;
            using difference_type = ptrdiff_t;
            using pointer = T*;
            using const_pointer = const T*;
            using reference = T&;
            using const_reference = const T&;
            template<class U> struct rebind { using other = allocator<U>; };

            T* address(T& x) const noexcept;
            const T* address(const T& x) const noexcept;

            T* allocate(size_t n, const void* hint);

            template<class U, class... Args>
                void construct(U* p, Args&&... args);
            template<class U>
                void destroy(U* p);

            size_t max_size() const noexcept;
    };

    T* address(T& x) const noexcept;
    const T* address(const T& x) const noexcept;

    Returns: addressof(x).

    T* allocate(size_t n, const void* hint);

    Returns: A pointer to the initial element of an array of storage of size n * sizeof(T), aligned appropriately for objects of type T. It is implementation-defined whether over-aligned types are supported (6.6.5).

    Remarks: The storage is obtained by calling ::operator new(std::size_t) (21.6.2), but it is unspecified when or how often this function is called.

    Throws: bad_alloc if the storage cannot be obtained.
template<class U, class... Args>
void construct(U* p, Args&&... args);

Effects: As if by: \texttt{:\textasciinewline:new((void *)p) U(std::forward<Args>(args)...)};

template<class U>
void destroy(U* p);

Effects: As if by \texttt{p->~U()}.\

size_t max_size() const noexcept;

Returns: The largest value \( N \) for which the call \texttt{allocate(N, 0)} might succeed.

D.11 Raw storage iterator

The header \texttt{<memory>} has the following addition:

namespace std {
    template<class OutputIterator, class T>
    class raw_storage_iterator {
        public:
            using iterator_category = output_iterator_tag;
            using value_type = void;
            using difference_type = void;
            using pointer = void;
            using reference = void;

            explicit raw_storage_iterator(OutputIterator x);

            raw_storage_iterator& operator*();
            raw_storage_iterator& operator=(const T& element);
            raw_storage_iterator& operator=(T&& element);
            raw_storage_iterator& operator++();
            OutputIterator base() const;
    };
}

\texttt{raw_storage_iterator} is provided to enable algorithms to store their results into uninitialized memory. The template parameter \texttt{OutputIterator} is required to have its \texttt{operator*} return an object for which \texttt{operator\&} is defined and returns a pointer to \texttt{T}, and is also required to satisfy the requirements of an output iterator (27.2.4).

explicit raw_storage_iterator(OutputIterator x);

Effects: Initializes the iterator to point to the same value to which \texttt{x} points.

raw_storage_iterator& operator*();

Returns: \texttt{*this}

raw_storage_iterator& operator=(const T& element);

Requires: \texttt{T} shall be \texttt{CopyConstructible}.

Effects: Constructs a value from \texttt{element} at the location to which the iterator points.

Returns: A reference to the iterator.

raw_storage_iterator& operator=(T&& element);

Requires: \texttt{T} shall be \texttt{MoveConstructible}.

Effects: Constructs a value from \texttt{std::move(element)} at the location to which the iterator points.

Returns: A reference to the iterator.

raw_storage_iterator& operator++();

Effects: Pre-increment: advances the iterator and returns a reference to the updated iterator.
raw_storage_iterator operator++(int);

Effects: Post-increment: advances the iterator and returns the old value of the iterator.

OutputIterator base() const;

Returns: An iterator of type OutputIterator that points to the same value as *this points to.

D.12 Temporary buffers [depr.temporary.buffer]

The header <memory> has the following additions:

```cpp
namespace std {
    template<class T>
    pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n) noexcept;
    template<class T>
    void return_temporary_buffer(T* p);
}
```

Effects: Obtains a pointer to uninitialized, contiguous storage for \( N \) adjacent objects of type \( T \), for some non-negative number \( N \). It is implementation-defined whether over-aligned types are supported (6.6.5).

Remarks: Calling get_temporary_buffer with a positive number \( n \) is a non-binding request to return storage for \( n \) objects of type \( T \). In this case, an implementation is permitted to return instead storage for a non-negative number \( N \) of such objects, where \( N \neq n \) (including \( N = 0 \)). [Note: The request is non-binding to allow latitude for implementation-specific optimizations of its memory management. —end note]

Returns: If \( n \leq 0 \) or if no storage could be obtained, returns a pair \( P \) such that \( P.first \) is a null pointer value and \( P.second = 0 \); otherwise returns a pair \( P \) such that \( P.first \) refers to the address of the uninitialized storage and \( P.second \) refers to its capacity \( N \) (in the units of \( \text{sizeof}(T) \)).

```cpp
template<class T> void return_temporary_buffer(T* p);
```

Effects: Deallocation of the storage referenced by \( p \).

Requires: \( p \) shall be a pointer value returned by an earlier call to get_temporary_buffer that has not been invalidated by an intervening call to return_temporary_buffer(T*).

Throws: Nothing.

D.13 Deprecated type traits [depr.meta.types]

The header <type_traits> has the following addition:

```cpp
namespace std {
    template<class T> struct is_literal_type;
    template<class T> constexpr bool is_literal_type_v = is_literal_type<T>::value;

    template<class> struct result_of; // not defined
    template<class Fn, class... ArgTypes> struct result_of<Fn(ArgTypes...)>;
    template<class T> using result_of_t = typename result_of<T>::type;

    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.

```cpp
template<class T> struct is_literal_type;
```

Requires: remove_all_extents_t<T> shall be a complete type or cv void.

is_literal_type<T> is a UnaryTypeTrait (23.15.1) with a base characteristic of true_type if \( T \) is a literal type (6.7), and false_type otherwise.
template<class Fn, class... ArgTypes> struct result_of<Fn(ArgTypes...)>;

Requires: Fn and all types in the parameter pack ArgTypes shall be complete types, cv void, or arrays of unknown bound.

The partial specialization result_of<Fn(ArgTypes...)> is a TransformationTrait (23.15.1) whose member typedef type is defined if and only if invoke_result<Fn, ArgTypes...>::type (23.14.4) is defined. If type is defined, it names the same type as invoke_result_t<Fn, ArgTypes...>.

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 UnaryTypeTrait (23.15.1) 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 (8.4.5.1) is a POD type.—end note]

D.14 Deprecated iterator primitives [depr.iterator.primitives]

D.14.1 Basic iterator [depr.iterator.basic]

The header <iterator> has the following addition:

```cpp
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:

```cpp
class MyIterator :
    public iterator<bidirectional_iterator_tag, double, long, T*, T&> {
        // code implementing ++, etc.
    };
```
—end example]

D.15 Deprecated shared_ptr observers [depr.util.smartptr.shared.obs]

The following member is defined in addition to those members specified in 23.11.3:

```cpp
namespace std {
    template<class T> class shared_ptr {
        public:
            bool unique() const noexcept;
    };
}
```

bool unique() const noexcept;

Returns: use_count() == 1.

D.16 Deprecated shared_ptr atomic access [depr.util.smartptr.shared.atomic]

The header <memory> has the following additions:
namespace std {
  
  template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);

  template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);

  template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

  template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);

  template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);

  template<class T>
  shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
  bool atomic_compare_exchange_weak(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

  template<class T>
  bool atomic_compare_exchange_strong(shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);

  template<class T>
  bool atomic_compare_exchange_weak_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure);

  template<class T>
  bool atomic_compare_exchange_strong_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w, memory_order success, memory_order failure);
}

Concurrent access to a shared_ptr object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this subclause and the instance is passed as their first argument.

The meaning of the arguments of type memory_order is explained in 32.4.

  template<class T> bool atomic_is_lock_free(const shared_ptr<T>* p);

Requires: p shall not be null.

Returns: true if atomic access to *p is lock-free, false otherwise.

Throws: Nothing.

  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 or 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 32.6.1.

D.17 Deprecated standard code conversion facets [depr.locale.stdcvt]

The header <codecvt> provides code conversion facets for various character encodings.
D.17.1  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.17.2  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 UCS2 or UCS4 (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 UCS2 or UCS4 (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:
The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.

Endianness shall not affect how multibyte sequences are read or written.

The multibyte sequences may be written as either a text or a binary file.

See also: ISO/IEC 10646-1:1993.

D.18 Deprecated convenience conversion interfaces

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;
}
```

D.18.1 Class template `wstring_convert`

Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales.

Example: If you want to use the code conversion facet `codecvt_utf8` to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don’t want to alter the locale for `cout`, you can write something like:

```cpp
wstring_convert<std::codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
```

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;

      explicit wstring_convert(Codecvt* pcvt = new Codecvt);
      wstring_convert(Codecvt* pcvt, state_type state);
      explicit wstring_convert(const byte_string& byte_err,
                              const wide_string& wide_err = wide_string());
      ~wstring_convert();

      wstring_convert(const wstring_convert&) = delete;
      wstring_convert& operator=(const wstring_convert&) = delete;

      wide_string from_bytes(char byte);
      wide_string from_bytes(const char* ptr);
      wide_string from_bytes(const byte_string& str);
      wide_string from_bytes(const char* first, const char* last);

      byte_string to_bytes(Elem wchar);
      byte_string to_bytes(const Elem* wptr);
      byte_string to_bytes(const wide_string& wstr);
      byte_string to_bytes(const Elem* first, const Elem* last);
  }
}
size_t converted() const noexcept;
state_type state() const;

private:
  byte_string byte_err_string; // exposition only
  wide_string wide_err_string; // exposition only
  Codecvt* cvtptr; // exposition only
  state_type cvtstate;   // exposition only
  size_t cvtcount;      // exposition only
};

The class template describes an object that controls conversions between wide string objects of class `basic_string<Elem, char_traits<Elem>, 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>`.

An object of this class template stores:

1. `byte_err_string` — a byte string to display on errors
2. `wide_err_string` — a wide string to display on errors
3. `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)
4. `cvtstate` — a conversion state object
5. `cvtcount` — a conversion count

```cpp
using byte_string = basic_string<char, char_traits<char>, Byte_alloc>;
```

The type shall be a synonym for `basic_string<char, char_traits<char>, Byte_alloc>`. `size_t converted() const noexcept;
Returns: cvtcount.

```cpp
wide_string from_bytes(char byte);
wide_string from_bytes(const char* ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char* first, const char* last);
```

Effects: The first member function shall convert the single-element sequence `byte` to a wide string. The second member function shall convert the null-terminated sequence beginning at `ptr` to a wide string. The third member function shall convert the sequence stored in `str` to a wide string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a wide string.

In all cases:

1. If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.
2. The number of input elements successfully converted shall be stored in `cvtcount`.

Returns: If no conversion error occurs, the member function shall return the converted wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class `range_error`.

```cpp
using int_type = typename wide_string::traits_type::int_type;
```

The type shall be a synonym for `wide_string::traits_type::int_type`. `state_type state() const;
returns cvtstate.
```

```cpp
using state_type = typename Codecvt::state_type;
```

The type shall be a synonym for `Codecvt::state_type`.

§ D.18.1 1325
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem* wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem* first, const Elem* last);

**Effects:** The first member function shall convert the single-element sequence `wchar` to a byte string. The second member function shall convert the null-terminated sequence beginning at `wptr` to a byte string. The third member function shall convert the sequence stored in `wstr` to a byte string. The fourth member function shall convert the sequence defined by the range `[first, last)` to a byte string.

In all cases:
- If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.
- The number of input elements successfully converted shall be stored in `cvtcount`.

**Returns:** If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string. Otherwise, the member function shall throw an object of class `range_error`.

using wide_string = basic_string<Elem, char_traits<Elem>, Wide_alloc>;

The type shall be a synonym for `basic_string<Elem, char_traits<Elem>, Wide_alloc>`.

explicit wstring_convert(Codecvt* pcvt = new Codecvt);
wstring_convert(Codecvt* pcvt, state_type state);
explicit wstring_convert(const byte_string& byte_err, const wide_string& wide_err = wide_string());

**Requires:** For the first and second constructors, `pcvt` ! = `nullptr`.

**Effects:** The first constructor shall store `pcvt` in `cvtptr` and default values in `cvtstate`, `byte_err_string`, and `wide_err_string`. The second constructor shall store `pcvt` in `cvtptr`, `state` in `cvtstate`, and default values in `byte_err_string` and `wide_err_string`; moreover the stored state shall be retained between calls to `from_bytes` and `to_bytes`. The third constructor shall store new `Codecvt` in `cvtptr`, `state_type()` in `cvtstate`, `byte_err` in `byte_err_string`, and `wide_err` in `wide_err_string`.

~wstring_convert();

**Effects:** The destructor shall delete `cvtptr`.

### D.18.2 Class template wbuffer_convert

Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

```cpp
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;

            explicit wbuffer_convert(streambuf* bytebuf = nullptr, Codecvt* pcvt = new Codecvt, state_type state = state_type());
            ~wbuffer_convert();

            wbuffer_convert(const wbuffer_convert&) = delete;
            wbuffer_convert& operator=(const wbuffer_convert&) = delete;

            streambuf* rdbuf() const;
            streambuf* rdbuf(streambuf* bytebuf);
            state_type state() const;

    };
}
```
private:
    streambuf* bufptr;  // exposition only
    Codecvt* cvtptr;    // exposition only
    state_type cvtstate; // exposition only
};

The class template describes a stream buffer that controls the transmission of elements of type Elem, whose character traits are described by the class Tr, to and from a byte stream buffer of type streambuf. Conversion between a sequence of Elem values and multibyte sequences is performed by an object of class Codecvt, which shall meet the requirements of the standard code-conversion facet codecvt<Elem, char, mbstate_t>.

An object of this class template stores:

(3.1) — bufptr — a pointer to its underlying byte stream buffer
(3.2) — cvtptr — a pointer to the allocated conversion object (which is freed when the wbuffer_convert object is destroyed)
(3.3) — cvtstate — a conversion state object

state_type state() const;
    Returns: cvtstate.

streambuf* rdbuf() const;
    Returns: bufptr.

streambuf* rdbuf(streambuf* bytebuf);
    Effects: Stores bytebuf in bufptr.
    Returns: The previous value of bufptr.

using state_type = typename Codecvt::state_type;

The type shall be a synonym for Codecvt::state_type.

explicit wbuffer_convert(
    streambuf* bytebuf = nullptr,
    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.
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 \textit{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 (29.8.2) 1008
adjacent.difference (29.8.11) 1015
adjustfield.manip (30.5.6.2) 1050
alg.3way (28.7.11) 937
alg.adjacent.find (28.5.8) 908
alg.all_of (28.5.1) 905
alg.any_of (28.5.2) 905
alg.binary.search (28.7.3) 925
alg.c.library (28.8) 938
alg.clamp (28.7.9) 936
alg.copy (28.6.1) 912
alg.count (28.5.9) 908
alg.equal (28.5.11) 909
alg.fill (28.6.6) 916
alg.find (28.5.5) 906
alg.find.end (28.5.6) 907
alg.find.first.of (28.5.7) 907
alg.foreach (28.5.4) 905
alg.generate (28.6.7) 916
alg.heap.operations (28.7.7) 932
alg.is_permutation (28.5.12) 910
alg.lex.comparison (28.7.10) 936
alg.merge (28.7.5) 928
alg.min.max (28.7.8) 934
alg.modifying.operations (28.6) 912
alg.move (28.6.2) 913
alg.none.of (28.5.3) 905
alg.nonmodifying (28.5) 905
alg.nth.element (28.7.2) 924
alg.partitions (28.7.4) 926
alg.permutation.generators (28.7.12) 938
alg.random.sample (28.6.12) 920
alg.random.shuffle (28.6.13) 921
alg.remove (28.6.8) 917
alg.replace (28.6.5) 915
alg.reverse (28.6.10) 919
alg.rotate (28.6.11) 919
alg.search (28.5.13) 911
alg.set.operations (28.7.6) 929
alg.sort (28.7.1) 922
alg.sorting (28.7) 921
alg.swap (28.6.3) 914
alg.transform (28.6.4) 914
alg.unique (28.6.9) 918
algorithm.stable (20.5.5.7) 432
algorithm.syn (28.2) 884
algorithms (Clause 28) 884
algorithms.general (28.1) 884
algorithms.parallel (28.4) 902
algorithms.parallel.defns (28.4.1) 902
algorithms.parallel.exceptions (28.4.4) 904
algorithms.parallel.exec (28.4.3) 903
algorithms.parallel.overloads (28.4.5) 904
algorithms.parallel.user (28.4.2) 903
algorithms.requirements (28.3) 901
alloc.errors (21.6.3) 453
allocator.adaptor (23.13) 588
allocator.adaptor.cnstr (23.13.3) 590
allocator.adaptor.members (23.13.4) 591
allocator.adaptor.syn (23.13.1) 588
allocator.adaptor.types (23.13.2) 590
allocator.globals (23.10.10.2) 550
allocator.members (23.10.10.1) 549
allocator.requirements (20.5.3.5) 422
allocator.requirements.completeness (20.5.3.5.1) 426
allocator.tag (23.10.7) 546
allocator.traits (23.10.9) 547
allocator.traits.members (23.10.9.2) 548
allocator.traits.types (23.10.9.1) 548
allocator.uses (23.10.8) 546
allocator.uses.construction (23.10.8.2) 547
allocator.uses.trait (23.10.8.1) 546
alt.headers (20.5.4.4) 429
any (23.8) 528
any.assign (23.8.3.2) 531
any.bad_any_cast (23.8.2) 529
any.class (23.8.3) 529
any.cons (23.8.3.1) 530
any.modifiers (23.8.3.3) 531
any.nonmembers (23.8.4) 532
any.observers (23.8.3.4) 532
any.synop (23.8.1) 529
arithmetic.operations (23.14.6) 598
arithmetic.operations.divides (23.14.6.4) 599
arithmetic.operations.minus (23.14.6.2) 598
arithmetic.operations.modulus (23.14.6.5) 599
arithmetic.operations.multiply (23.14.6.3) 599
arithmetic.operations.negative (23.14.6.6) 600
arithmetic.operations.plus (23.14.6.1) 598
array (26.3.7) 785
array.cons (26.3.7.2) 786
array.members (26.3.7.3) 787
array.overview (26.3.7.1) 785
array.special (26.3.7.4) 787
array.syn (26.3.2) 783
array.tuple (26.3.7.6) 787
array.zero (26.3.7.5) 787
assertions (22.3) 475
assertions.assert (22.3.2) 475
Cross references

1332
<table>
<thead>
<tr>
<th>Cross references</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator.container (27.8)</td>
<td>883</td>
</tr>
<tr>
<td>iterator.iterators (27.2.2)</td>
<td>856</td>
</tr>
<tr>
<td>iterator.operations (27.4.3)</td>
<td>865</td>
</tr>
<tr>
<td>iterator.primitives (27.4)</td>
<td>863</td>
</tr>
<tr>
<td>iterator.range (27.7)</td>
<td>882</td>
</tr>
<tr>
<td>iterator.requirements (27.2)</td>
<td>855</td>
</tr>
<tr>
<td>iterator.requirements.general (27.2.1)</td>
<td>855</td>
</tr>
<tr>
<td>iterator.synopsis (27.3)</td>
<td>860</td>
</tr>
<tr>
<td>iterators (Clause 27)</td>
<td>855</td>
</tr>
<tr>
<td>iterators.general (27.1)</td>
<td>855</td>
</tr>
<tr>
<td>language.support (Clause 21)</td>
<td>434</td>
</tr>
<tr>
<td>length.error (22.2.5)</td>
<td>473</td>
</tr>
<tr>
<td>lex (Clause 5)</td>
<td>9</td>
</tr>
<tr>
<td>lex.bool (5.13.6)</td>
<td>21</td>
</tr>
<tr>
<td>lex.ccon (5.13.3)</td>
<td>17</td>
</tr>
<tr>
<td>lex.charset (5.3)</td>
<td>10</td>
</tr>
<tr>
<td>lex.comment (5.7)</td>
<td>12</td>
</tr>
<tr>
<td>lex.digraph (5.5)</td>
<td>12</td>
</tr>
<tr>
<td>lex.ext (5.13.8)</td>
<td>22</td>
</tr>
<tr>
<td>lex.fcon (5.13.4)</td>
<td>18</td>
</tr>
<tr>
<td>lex.header (5.8)</td>
<td>12</td>
</tr>
<tr>
<td>lex.icon (5.13.2)</td>
<td>15</td>
</tr>
<tr>
<td>lex.key (5.11)</td>
<td>14</td>
</tr>
<tr>
<td>lex.literal (5.13)</td>
<td>15</td>
</tr>
<tr>
<td>lex.literal.kinds (5.13.1)</td>
<td>15</td>
</tr>
<tr>
<td>lex.name (5.10)</td>
<td>13</td>
</tr>
<tr>
<td>lex.nullptr (5.13.7)</td>
<td>21</td>
</tr>
<tr>
<td>lex.operators (5.12)</td>
<td>15</td>
</tr>
<tr>
<td>lex.phases (5.2)</td>
<td>9</td>
</tr>
<tr>
<td>lex.pnumber (5.9)</td>
<td>13</td>
</tr>
<tr>
<td>lex.pptoken (5.4)</td>
<td>11</td>
</tr>
<tr>
<td>lex.pptoken (5.4.3.2)</td>
<td>428</td>
</tr>
<tr>
<td>locale.collate.virtuals (25.4.4.1.2)</td>
<td>734</td>
</tr>
<tr>
<td>locale.cons (25.3.1.2)</td>
<td>713</td>
</tr>
<tr>
<td>locale.convenience (25.3.3)</td>
<td>715</td>
</tr>
<tr>
<td>locale.ctype (25.4.1.1)</td>
<td>716</td>
</tr>
<tr>
<td>locale.ctype.bname (25.4.1.2)</td>
<td>719</td>
</tr>
<tr>
<td>locale.ctype.members (25.4.1.1.1)</td>
<td>717</td>
</tr>
<tr>
<td>locale.ctype.virtuals (25.4.1.1.2)</td>
<td>717</td>
</tr>
<tr>
<td>locale.facet (25.3.1.1.2)</td>
<td>712</td>
</tr>
<tr>
<td>locale.global.templates (25.3.2)</td>
<td>715</td>
</tr>
<tr>
<td>locale.id (25.3.1.1.3)</td>
<td>713</td>
</tr>
<tr>
<td>locale.members (25.3.1.3)</td>
<td>714</td>
</tr>
<tr>
<td>locale.messages (25.4.7.1)</td>
<td>745</td>
</tr>
<tr>
<td>locale.messages.bname (25.4.7.2)</td>
<td>746</td>
</tr>
<tr>
<td>locale.messages.members (25.4.7.1.1)</td>
<td>745</td>
</tr>
<tr>
<td>locale.messages.virtuals (25.4.7.1.2)</td>
<td>745</td>
</tr>
<tr>
<td>locale.money.get (25.4.6.1)</td>
<td>740</td>
</tr>
<tr>
<td>locale.money.get.members (25.4.6.1.1)</td>
<td>740</td>
</tr>
<tr>
<td>locale.money.get.virtuals (25.4.6.1.2)</td>
<td>740</td>
</tr>
<tr>
<td>locale.money.put (25.4.6.2)</td>
<td>741</td>
</tr>
<tr>
<td>locale.money.put.members (25.4.6.2.1)</td>
<td>742</td>
</tr>
<tr>
<td>locale.money.put.virtuals (25.4.6.2.2)</td>
<td>742</td>
</tr>
<tr>
<td>locale.moneypoint (25.4.6.3)</td>
<td>742</td>
</tr>
<tr>
<td>locale.moneypoint.bname (25.4.6.4)</td>
<td>744</td>
</tr>
<tr>
<td>locale.moneypoint.members (25.4.6.3.1)</td>
<td>744</td>
</tr>
<tr>
<td>locale.moneypoint.virtuals (25.4.6.3.2)</td>
<td>744</td>
</tr>
<tr>
<td>locale.mm.put (25.4.2.2)</td>
<td>729</td>
</tr>
<tr>
<td>locale.num.get (25.4.2.1)</td>
<td>725</td>
</tr>
<tr>
<td>locale.num.get.members (25.4.3.1)</td>
<td>732</td>
</tr>
<tr>
<td>locale.num.get.virtuals (25.4.3.2)</td>
<td>733</td>
</tr>
<tr>
<td>locale.ops (25.3.1.4)</td>
<td>714</td>
</tr>
<tr>
<td>locale.statics (25.3.1.5)</td>
<td>715</td>
</tr>
<tr>
<td>locale.stat (25.2)</td>
<td>708</td>
</tr>
<tr>
<td>locale.time.get (25.4.5.1)</td>
<td>735</td>
</tr>
<tr>
<td>locale.time.get.bname (25.4.5.2)</td>
<td>738</td>
</tr>
<tr>
<td>locale.time.get.members (25.4.5.1.1)</td>
<td>736</td>
</tr>
<tr>
<td>locale.time.get.virtuals (25.4.5.1.2)</td>
<td>737</td>
</tr>
<tr>
<td>locale.time.put (25.4.5.3)</td>
<td>738</td>
</tr>
<tr>
<td>locale.time.put.bname (25.4.5.4)</td>
<td>739</td>
</tr>
<tr>
<td>locale.time.put.members (25.4.5.3.1)</td>
<td>739</td>
</tr>
<tr>
<td>locale.time.put.virtuals (25.4.5.3.2)</td>
<td>739</td>
</tr>
<tr>
<td>locale.types (25.3.1.1)</td>
<td>711</td>
</tr>
<tr>
<td>locales (25.3)</td>
<td>709</td>
</tr>
<tr>
<td>localization (Clause 25)</td>
<td>708</td>
</tr>
<tr>
<td>localization.general (25.1)</td>
<td>708</td>
</tr>
<tr>
<td>logic.error (22.2.2)</td>
<td>472</td>
</tr>
<tr>
<td>logical.conversions (23.14.8)</td>
<td>602</td>
</tr>
<tr>
<td>logical.conversions.and (23.14.8.1)</td>
<td>602</td>
</tr>
<tr>
<td>logical.conversions.not (23.14.8.3)</td>
<td>603</td>
</tr>
<tr>
<td>logical.conversions.or (23.14.8.2)</td>
<td>603</td>
</tr>
<tr>
<td>lower.bound (28.7.3.1)</td>
<td>925</td>
</tr>
<tr>
<td>macro.names (20.5.4.3.2)</td>
<td>428</td>
</tr>
<tr>
<td>make.heap (28.7.7.3)</td>
<td>933</td>
</tr>
<tr>
<td>map (26.4.4)</td>
<td>812</td>
</tr>
<tr>
<td>map.access (26.4.4.3)</td>
<td>815</td>
</tr>
<tr>
<td>map.cons (26.4.4.2)</td>
<td>815</td>
</tr>
<tr>
<td>map.modifiers (26.4.4.4)</td>
<td>815</td>
</tr>
<tr>
<td>map.overview (26.4.4.1)</td>
<td>812</td>
</tr>
<tr>
<td>map.special (26.4.4.5)</td>
<td>817</td>
</tr>
</tbody>
</table>
Cross references 1339
<table>
<thead>
<tr>
<th>Cross references 1345</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread.lock.algorithm (33.4.5)</td>
</tr>
<tr>
<td>thread.lock.guard (33.4.4.1)</td>
</tr>
<tr>
<td>thread.lockScoped (33.4.4.2)</td>
</tr>
<tr>
<td>thread.lock.shared (33.4.4.4)</td>
</tr>
<tr>
<td>thread.lock.shared.cons (33.4.4.4.1)</td>
</tr>
<tr>
<td>thread.lock.shared.locking (33.4.4.4.2)</td>
</tr>
<tr>
<td>thread.lock.shared.mod (33.4.4.4.3)</td>
</tr>
<tr>
<td>thread.lock.shared.obs (33.4.4.4.4)</td>
</tr>
<tr>
<td>thread.lock.unique (33.4.4.3)</td>
</tr>
<tr>
<td>thread.lock.unique.cons (33.4.4.3.1)</td>
</tr>
<tr>
<td>thread.lock.unique.locking (33.4.4.3.2)</td>
</tr>
<tr>
<td>thread.lock.unique.mod (33.4.4.3.3)</td>
</tr>
<tr>
<td>thread.lock.unique.obs (33.4.4.3.4)</td>
</tr>
<tr>
<td>thread.mutex (33.4)</td>
</tr>
<tr>
<td>thread.mutex.class (33.4.3.2.1)</td>
</tr>
<tr>
<td>thread.mutex.recursive (33.4.3.2.2)</td>
</tr>
<tr>
<td>thread.mutex.requirements (33.4.3)</td>
</tr>
<tr>
<td>thread.mutex.requirements.general (33.4.3.1)</td>
</tr>
<tr>
<td>thread.mutex.requirements.mutex (33.4.3.2)</td>
</tr>
<tr>
<td>thread.once (33.4.6)</td>
</tr>
<tr>
<td>thread.once.callonce (33.4.6.2)</td>
</tr>
<tr>
<td>thread.once.onceflag (33.4.6.1)</td>
</tr>
<tr>
<td>thread.req (33.2)</td>
</tr>
<tr>
<td>thread.req.exception (33.2.2)</td>
</tr>
<tr>
<td>thread.req.lockable (33.2.5)</td>
</tr>
<tr>
<td>thread.req.lockable.basic (33.2.5.2)</td>
</tr>
<tr>
<td>thread.req.lockable.generic (33.2.5.1)</td>
</tr>
<tr>
<td>thread.req.lockable.req (33.2.5.3)</td>
</tr>
<tr>
<td>thread.req.lockable.timed (33.2.5.4)</td>
</tr>
<tr>
<td>thread.req.native (33.2.3)</td>
</tr>
<tr>
<td>thread.req.paramname (33.2.1)</td>
</tr>
<tr>
<td>thread.req.timing (33.2.4)</td>
</tr>
<tr>
<td>thread.sharedmutex.class (33.4.3.4.1)</td>
</tr>
<tr>
<td>thread.sharedmutex.requirements (33.4.3.4)</td>
</tr>
<tr>
<td>thread.sharedtimedmutex.class (33.4.3.5.1)</td>
</tr>
<tr>
<td>thread.sharedtimedmutex.requirements (33.4.3.5)</td>
</tr>
<tr>
<td>thread.syn (33.3.1)</td>
</tr>
<tr>
<td>thread.thread.algorithm (33.3.2.7)</td>
</tr>
<tr>
<td>thread.thread.assign (33.3.2.4)</td>
</tr>
<tr>
<td>thread.thread.class (33.3.2)</td>
</tr>
<tr>
<td>thread.thread.constr (33.3.2.2)</td>
</tr>
<tr>
<td>thread.thread.destr (33.3.2.3)</td>
</tr>
<tr>
<td>thread.thread.id (33.3.2.1)</td>
</tr>
<tr>
<td>thread.thread.member (33.3.2.5)</td>
</tr>
<tr>
<td>thread.thread.static (33.3.2.6)</td>
</tr>
<tr>
<td>thread.thread.this (33.3.3)</td>
</tr>
<tr>
<td>thread.tickets (33.3)</td>
</tr>
<tr>
<td>thread.timedmutex.class (33.4.3.3.1)</td>
</tr>
<tr>
<td>thread.timedmutex.recursive (33.4.3.3.2)</td>
</tr>
<tr>
<td>thread.timedmutex.requirements (33.4.3.3)</td>
</tr>
</tbody>
</table>

- time (23.17) 639
- time.clock (23.17.7) 652
- time.clock.hires (23.17.7.3) 653
- time.clock.req (23.17.3) 641
- time.clock.steady (23.17.7.2) 652
- time.clock.system (23.17.7.1) 652
- time.duration (23.17.5) 643
- time.duration.alg (23.17.5.9) 649
- time.duration.arithmetic (23.17.5.3) 645
- time.duration.cast (23.17.5.7) 647
- time.duration.comparisons (23.17.5.6) 647
- time.duration.cons (23.17.5.1) 644
- time.duration.literals (23.17.5.8) 648
- time.duration.nonmember (23.17.5.5) 646
- time.duration.observer (23.17.5.2) 645
- time.duration.special (23.17.5.4) 646
- time.general (23.17.1) 639
- time.point (23.17.6) 649
- time.point.arithmetic (23.17.6.3) 650
- time.point.cast (23.17.6.7) 651
- time.point.comparisons (23.17.6.6) 651
- time.point.cons (23.17.6.1) 650
- time.point.nonmember (23.17.6.5) 650
- time.point.observer (23.17.6.2) 650
- time.point.special (23.17.6.4) 650
- time.syn (23.17.2) 639
- time.trait (23.17.4) 642
- time.trait.duration_values (23.17.4.2) 642
- time.trait.is_fp (23.17.4.1) 642
- transform.exclusive.scan (29.8.9) 1013
- transform.inclusive.scan (29.8.10) 1014
- transform.reduce (29.8.5) 1010
- tuple (23.5) 495
- tuple.apply (23.5.3.5) 502
- tuple.assign (23.5.3.2) 500
- tuple.cnstr (23.5.3.1) 498
- tuple.creation (23.5.3.4) 501
- tuple.element (23.5.3.7) 503
- tuple.general (23.5.1) 495
- tuple.helper (23.5.3.6) 503
- tuple.rel (23.5.3.8) 504
- tuple.special (23.5.3.10) 505
- tuple.swap (23.5.3.3) 501
- tuple.syn (23.5.2) 496
- tuple.trait (23.5.3.9) 505
- tuple.tuple (23.5.3) 497
- type.descriptions (20.4.2.1) 412
- type.descriptions.general (20.4.2.1.1) 412
- type.index (23.18) 653
- type.index.hash (23.18.4) 654
- type.index.members (23.18.3) 654
- type.index.overview (23.18.2) 653
- type.index.synopsis (23.18.1) 653
- type.info (21.7.2) 456
- typeinfo.syn (21.7.1) 455
- uncaught.exceptions (21.8.5) 459
- underflow.error (22.2.10) 475
- uninitialized.construct.default (23.10.11.2) 550
- uninitialized.construct.value (23.10.11.3) 550
- uninitialized.copy (23.10.11.4) 551
- uninitialized.fill (23.10.11.6) 551
- uninitialized.move (23.10.11.5) 551
- unique.ptr (23.11.1) 552
- unique.ptr.create (23.11.1.4) 560
variant.variant (23.7.3)  519
variant.visit (23.7.7)  527
vector (26.3.11)  803
vector.bool (26.3.12) 807
vector.capacity (26.3.11.3)  805
vector.cons (26.3.11.2)  805
vector.data (26.3.11.4)  806
vector.modifiers (26.3.11.5)  806
vector.overview (26.3.11.1)  803
vector.special (26.3.11.6)  807
vector.syn (26.3.6)  785

wide.stream.objects (30.4.4)  1036

zombie.names (20.5.4.3.1)  427
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.

array.data see array.members
array.fill see array.members
array.size see array.members
array.swap see array.members

basic.scope.proto see basic.scope.param

fs.definitions see fs.class.path, fs.conform.os,
fs.general, fs.path.fmt.cvt,
fs.path.generic, fs.race.behavior

util.smartptr removed
utility.from.chars see charconv.from.chars
utility.to.chars see charconv.to.chars

variant.traits 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, unary minus, see operator, subtraction
-> see operator, class member access
->*, see operator, pointer to member
-- see operator, decrement
., see operator, class member access
.* see operator, pointer to member
..., see ellipsis
/, see operator, division
:
    bit-field declaration, 222
    label specifier, 130
:: see operator, scope resolution
::* see operator, pointer-to-member
<, see operator, less than
    template and, 311, 312
<< see operator, left shift
<= see operator, less than or equal to
=> see operator, three-way comparison
= see assignment operator
== see equality operator
>, see operator, greater than
>= see operator, greater than or equal to
>>, see operator, right shift
?: see operator, conditional expression
[] see operator, subscripting, see declarator, array
# operator, 399, 400
## operator, 401
#define, 399
#elif, 397
#else, 397
#endif, 398
#error see preprocessing directives, error
#if, 397, 431
#endif, 397
#include, 398, 417
#line see preprocessing directives, line control
#pragma see preprocessing directives, pragma
#undef, 402, 428
% see operator, remainder
& see operator, address-of, see operator, bitwise
    AND, see declarator, reference
&& see operator, logical AND
~ see operator, bitwise exclusive OR
\ see backslash character
{} see block statement, 131
    class declaration, 211
    class definition, 211
equal to, see operator, equality
enum declaration, 156
initializer list, 199
_ see character, underscore
__cplusplus, 404
__DATE__, 404
__FILE__, 404
__has_include, 396
__LINE__, 405
__STDCPP_DEFAULT_NEW_ALIGNMENT__, 405
__STDCPP.StrictPointerType__ , 405
__STDCPP_THREADS__, 405
__STDC_HOSTED__, 405
__STDC_ISO_10646__, 405
__STDC_MB_MIGHT_NEQ_WC__, 405
__STDC_VERSION__, 405
__STDC__, 405
__TIME__, 405
__VA_ARGS__, 399, 400
__VA_OPT__, 399, 400
1 see operator, bitwise inclusive OR
11 see operator, logical OR
~ see operator, ones’ complement, see destructor
0 see also zero, null
    null character, see character, null
    string terminator, 21
abort, 74, 136
absolute path, 1114
abstract-declarator, 180, 1273
abstract-pack-declarator, 180, 1273
access, 3
access control, 238–246
anonymous union, 225
    base class, 240
    base class member, 227
    class member, 101
default, 238
default argument, 239
friend function, 242
member function and, 247
member name, 238
multiple access, 246
nested class, 246
overload resolution and, 230
overloading and, 279
private, 238
Index
move, see assignment operator, move, 409

assignment operator
copy, 247, 270–272
hidden, 271
implicitly declared, 270
implicitly defined, 271
inaccessible, 267
non-trivial, 271
trivial, 271
virtual bases and, 272
move, 247, 270–272
hidden, 271
implicitly declared, 270
implicitly defined, 271
inaccessible, 267
non-trivial, 271
trivial, 271
overloaded, 300
assignment-expression, 125, 1267
assignment-operator, 125, 1267
associated constraints, 320
associative containers
exception safety, 771
requirements, 771
unordered, see unordered associative containers
asynchronous provider, 1247
asynchronous return object, 1247
at least as constrained, 322
at least as specialized as, see more specialized
atexit, 73
<atomic>, 1198
atomic constraint, 320
identical, 320
atomic operations, see operation, atomic
atomic smart pointers, 575–579
attribute, 173–176
alignment, 175
carries dependency, 175
deprecated, 176
fallthrough, 177
maybe unused, 177
nodiscard, 178
noretur, 178
syntax and semantics, 173
attribute, 173, 1271
attribute-argument-clause, 174, 1272
attribute-declaration, 139, 1269
attribute-list, 173, 1271
attribute-namespace, 174, 1272
attribute-scoped-token, 174, 1272
attribute-specifier, 173, 1271
attribute-specifier-seq, 173, 1271
attribute-token, 173, 1272
attribute-using-prefix, 173, 1271
automatic storage duration, 54
awk, 1167
backslash character, 18
bad_alloc, 114
bad_cast, 103
bad_typeid, 104
balanced-token, 174, 1272
balanced-token-seq, 174, 1272
base characteristic, 614
base class, 227, 228
dependent, 349
direct, 227
indirect, 227
non-virtual, 228
overloading and, 278
private, 240
protected, 240
public, 240
virtual, 228
base class subobject, 49
base-clause, 227, 1275
base-specifier, 227, 1275
base-specifier-list, 227, 1275
begin
unordered associative containers, 781
behavior
conditionally-supported, 3, 6
default, 408, 409, 412
implementation-defined, 4, 6
locale-specific, 4
observable, 6, 7
on receipt of signal, 65
required, 410, 412
undefined, 5–7, 879
unspecified, 5, 7
Ben, 278
Bernoulli distributions, 972–975
bernoulli_distribution
discrete probability function, 972
Bessel functions
I_ν, 1027
J_ν, 1028
K_ν, 1028
N_ν, 1028
j_ν, 1030
n_ν, 1031
beta functions \( B \), 1026
binary fold, 95
binary left fold, 95
binary operator
interpretation of, 300
overloaded, 300
binary right fold, 95
binary-digit, 15, 1261
binary-exponent-part, 19, 1262
binary-literal, 15, 1261
BinaryTypeTrait, 615
bind directly, 206
binding
reference, 204
binomial_distribution
discrete probability function, 973
bit-field, 222
address of, 222
alignment of, 222
implementation-defined alignment of, 222
implementation-defined sign of, 1287
type of, 222
unnamed, 222
zero width of, 222

bitmask
empty, 413

<bitset>, 533
block, 3
initialization in, 137
block scope, 31
block statement, see statement, compound
block structure, 137
block with forward progress guarantee delegation, 70

block-declaration, 139, 1268
body
function, 192
Bond
James Bond, 93
Boolean, 222
Boolean literal, 21
boolean literal, see literal, boolean
Boolean type, 60
boolean-literal, 21, 1263
bound arguments, 607
bound, of array, 185
brace-or-equal-initializer, 196, 1273
braced-init-list, 196, 1273
brains
names that want to eat your, 427

bucket
unordered associative containers, 780

bucket_count
unordered associative containers, 780

bucket_size
unordered associative containers, 780

buckets, 772
byte, 48, 110

C
linkage to, 171
standard, 1
standard library, 2
c-char, 17, 1261
c-char-sequence, 17, 1261
call
operator function, 300
call signature, 596
call wrapper, 596
forwarding, 596
simple, 596
type, 596
callable object, 596
callable type, 596, 609
capture
implicit, 92
capture, 91, 1264
capture-default, 91, 1264
capture-list, 91, 1264
captured, 93
by copy, 94
by reference, 94
carries a dependency, 66
carry
subtract_with_carry_engine, 962
<cassert>, 417, 475, 748, 1301
cast
base class, 106
const, 107, 117
derived class, 106
dynamic, 103, 456
construction and, 267
destruction and, 267
integer to pointer, 107
lvalue, 105, 106
pointer to integer, 106
pointer-to-function, 107
pointer-to-member, 106, 107
reference, 105, 107
reinterpret, 106, 117
integer to pointer, 107
lvalue, 106
pointer to integer, 106
pointer-to-function, 107
pointer-to-member, 107
reference, 107
static, 105, 117
lvalue, 105
reference, 105
undefined pointer-to-function, 107
cast-expression, 117, 1266
casting, 101
casting away constness, 108
catch, 386
cats
interfering with canines, 455
cauchy_distribution
probability density function, 980
cbegin
unordered associative containers, 781
<ccomplex>, 1301, 1303
<cctype>, 704
cend
unordered associative containers, 781
cerrno>, 428, 476
<cfenv>, 941
char
implementation-defined sign of, 59
char-like object, 660
char-like type, 660
char16_t, see type, char16_t
char16_t character, 17
char32_t, see type, char32_t
char32_t character, 17

char_class_type
regular expression traits, 1159
character, 408
  decimal-point, 414
  multibyte, 4
  null, 10
signed, 59
  source file, 9
  underscore, 14
  in identifier, 13

character literal, see literal, character
character set, 10–11
  basic execution, 10, 48
  basic source, 9, 10
  execution, 10
character string, 20
character string literal, 401
character-literal, 17, 1261
<charconv>, 656
checking
  point of error, 343
  syntax, 343

chi_squared_distribution
probability density function, 980
<chrono>, 639
chunk, 584
<cinttypes>, 1156, 1157
<ciso646>, 1301
class, 61, 211–224
  abstract, 236
  associated, 38
  base, 429, 432
  cast to incomplete, 117
  constructor and abstract, 237
  definition, 26
  derived, 432
  linkage of, 46
  linkage specification, 172
  local, see local class
  member function, see member function, class
  nested, see nested class
  pointer to abstract, 237
  polymorphic, 232
  scope of enumerator, 158
  standard-layout, 212
  trivial, 211
  trivially copyable, 211
  union-like, 226
  unnamed, 144
  variant member of, 226
class name
  elaborated, 152, 214
  point of declaration, 214
  scope of, 213
typedef, 144, 214
class object
  assignment to, 125
  member, 216
  sizeof, 110
  class object copy, see constructor, copy
  class object initialization, see constructor
class-head, 211, 1274
class-head-name, 211, 1274
class-key, 211, 1274
class-name, 211, 1274
class-or-decltype, 227, 1275
class-specifier, 211, 1274
class-virt-specifier, 211, 1274

clear
  unordered associative containers, 780
<climits>, 1307
<locale>, 414, 1302
closure object, 87
closure type, 87
<cmath>, 1016, 1025
<codecvt>, 1322
coherence
  read-read, 68
  read-write, 68
  write-read, 68
  write-write, 68
collating element, 1158
comma operator, see operator, comma
comment, 11–12
  /* */, 12
  //, 12
common comparison type, 274
common initial sequence, 217
<compare>, 120, 462
compare-expression, 120, 1266
comparison
  pointer, 121
  pointer to function, 121
  undefined pointer, 119
comparison category types, 463
compatible with
  shared_ptr, 563
compilation
  separate, 9
  compiler control line, see preprocessing directives
  complete object, 49
  complete object of, 50
  completely defined, 216
<complex>, 942
<complex.h>, 1301
component, 408
composite pointer type, 82
compound-requirement, 97, 1265
compound-statement, 131, 1268
concatenation
  macro argument, see ## operator
  string, 21
concept, 341
  variadic, 341
concept-definition, 306, 1276
concept-name, 306, 1276
Index 1353
concurrent forward progress guarantees, 69
condition, 130, 1267
conditions
rules for, 130
<condition_variable>, 1239
conditional-expression
throw-expression in, 123
conditional-expression, 123, 1267
conditionally-supported behavior, see behavior, conditionally-supported
conditionally-supported-directive, 395, 1278
conflict, 66
conformance requirements, 6–7
class templates, 6
classes, 6
general, 6
library, 6
method of description, 6
conjunction, 319
consistency
linkage, 142
linkage specification, 172
type declaration, 48
cast, 62
cast away, 108
constructor and, 220, 248
destructor and, 220, 255
linkage of, 46
overloading and, 277
cost member function, 219
cost object
undefined change to, 149
cost volatile member function, 219
cost-default-constructible, 197
cost-qualified, 62
cost-volatile-qualified, 62
cost_cast, see cast, const
cost_local_iterator, 773
 unordered associative containers, 773
constant, 15, 84
enumeration, 157
null pointer, 78, 79
constant expression, 126
permitted result of, 129
constant initialization, 71
constant initializer, 71
constant iterator, 855
constant subexpression, 408
constant-expression, 126, 1267
constexpr function, 145
constexpr if, 132
constituent expression, 63
conststrained-parameter, 308, 1276
constraint, 319
associated, see associated constraints
normalization, 321
satisfaction
atomic, 320
conjunction, 319
disjunction, 319
subsumption, 322
constraint-expression, 320, 1276
constraint-logical-and-expression, 306, 1276
constraint-logical-or-expression, 306, 1276
construction, 265–267
dynamic cast and, 267
member access, 265
move, 409
pointer to member or base, 266
typeid operator, 267
virtual function call, 266
constructor, 247
address of, 249
array of class objects and, 259
converting, 253
copy, 247, 250, 267–270, 414
elision, 272
implicitly declared, 268
implicitly defined, 269
inaccessible, 267
nontrivial, 269
trivial, 269
default, 247, 248
non-trivial, 248
trivial, 248
exception handling, see exception handling, constructors and destructors
explicit call, 249
implicitly called, 248
implicitly defined, 248
inheritance of, 248
move, 247, 267–270
elision, 272
implicitly declared, 269
implicitly defined, 269
inaccessible, 267
non-trivial, 269
trivial, 269
non-trivial, 248
random number distribution requirement, 956
random number engine requirement, 953
union, 224
constructor, conversion by, see conversion, user-defined
contained value
any, 530
optional, 508
variant, 520
container
contiguous, 754
contains a value
optional, 508
context
non-deduced, 378
contextually converted constant expression of type bool, see conversion, contextual
contextually converted to bool, see conversion, contextual

Index 1354
contextually implicitly converted, 75
contiguous container, 754
contiguous iterators, 855
continue
and handler, 386
and try block, 386
control line, see preprocessing directives
control-line, 395, 1277
conventions, 412
lexical, 9–23
conversion
argument, 187
array-to-pointer, 76
bool, 78
boolean, 79
class, 252
contextual, 75
contextual to bool, 75
contextual to constant expression of type bool, 129
deduced return type of user-defined, 255
derived-to-base, 291
floating to integral, 78
floating-point, 78
function pointer, 79
function-to-pointer, 76
implementation-defined pointer integer, 106, 107
implicit, 75, 252
implicit user-defined, 252
inheritance of user-defined, 255
integer rank, 63
integral, 78
integral to floating, 78
lvalue-to-rvalue, 76, 1283
narrowing, 210
overload resolution and, 287
overload resolution and pointer, 299
pointer, 78
pointer-to-member, 79
void*, 79
qualification, 77
return type, 136
standard, 75–79
temporary materialization, 76
to signed, 78
to unsigned, 78
type of, 254
user-defined, 252–254
usual arithmetic, 83
virtual user-defined, 255
conversion explicit type, see casting
conversion function, see conversion, user-defined, 254
conversion rank, 292
conversion-declarator, 254, 1275
conversion-function-id, 254, 1275
conversion-type-id, 254, 1275
copy
class object, see constructor, copy, see
assignment operator, copy
copy constructor
random number engine requirement, 953
copy deduction candidate, 286
copy elision, see constructor, copy, elision
copy-initialization, 198
copy-list-initialization, 206
CopyInsertable into X, 754
count
unordered associative containers, 780
<csetjmp>, 428, 469, 470, 1302
<csignal>, 469, 470
<cstdalign>, 1301, 1303
<cstddarg>, 187, 428, 469
<cstdbool>, 1301, 1303
<cstddef>, 110, 119, 1301, 1302
<cstdint>, 446
<cstdio>, 1035, 1036, 1094, 1154, 1156, 1302
<cstl<lib>, 74, 417, 435, 447, 469, 552, 707, 938, 987, 1025, 1301, 1302, 1304
<cstring>, 414, 704, 1302, 1307, 1311
<ctgmath>, 1301, 1304
<ctime>, 469, 653, 709, 1302
tor-initializer, 260, 1275
<ctype.h>, 704
<cuchar>, 428, 706, 1301
current instantiation, 348
dependent member of the, 349
member of the, 349
currently handled exception, see exception
handling, currently handled exception
cv-decomposition, 77
cv-qualification signature, 77
cv-qualifier, 62
top-level, 62
cv-qualifier, 180, 1272
cv-qualifier-seq, 180, 1272
<cwchar>, 428, 705, 707, 1301, 1302
<cwctype>, 428, 704
d-char, 20, 1263
d-char-sequence, 20, 1263
DAG
multiple inheritance, 229
non-virtual base class, 229
virtual base class, 229
data member, see member, 215
static, 215
data race, 68
deadlock, 408
deallocation function
usual, 55
deallocation functions, 54
decay
array, see conversion, array to pointer
function, see conversion, function to pointer
DECAY_COPY, 1216
decimal-floating-literal, 18, 1262
Index
dependent member of the current instantiation, see current instantiation, dependent member of the current instantiation, dependent name, see name, dependent
<deque>, 783
dereferencing, see also indirection
derivation, see inheritance
derived class, 227–237
   most, see most derived class
derived object
   most, see most derived object
designated-initializer-clause, 196, 1274
designated-initializer-list, 196, 1274
designator, 196, 1274
destruction, 265–267
   dynamic cast and, 267
   member access, 265
   pointer to member or base, 266
   typeid operator, 267
   virtual function call, 266
destructor, 255, 414
   default, 255
   exception handling, see exception handling, constructors and destructors
   explicit call, 256
   implicit call, 256
   implicitly defined, 256
   non-trivial, 255
   program termination and, 256
   pure virtual, 256
   union, 224
   virtual, 256
diagnosable rules, 6
diagnostic message, see message, diagnostic
digit, 13, 1260
digit-sequence, 19, 1262
digraph, see token, alternative, 12
direct member, 215
direct-initialization, 198
direct-list-initialization, 206
direct-non-list-initialization, 409
directed acyclic graph, see DAG
directive, preprocessing, see preprocessing
directives
directory, 1109
directory-separator, 1117
discard
   random number engine requirement, 953
discard_block_engine
   generation algorithm, 964
   state, 964
   textual representation, 964
   transition algorithm, 964
discarded statement, 132
discarded-value expression, 83
discrete probability function
   bernoulli_distribution, 972
   binomial_distribution, 973
   discrete_distribution, 983
discrete_distribution
   discrete probability function, 983
   weights, 983
disjunction, 319
distribution, see random number distribution
dogs
   obliviousness to interference, 455
domain error, 1026
dominance
   virtual base class, 231
dot operator, see operator, class member access
   dynamic binding, see virtual function
   dynamic initialization, 71
   dynamic type, see type, dynamic
dynamic_cast, see cast, dynamic
E (complete elliptic integrals), 1027
E (incomplete elliptic integrals), 1029
ECMA-262, 2
ECMAScript, 1167, 1195
egrep, 1167
Ei (exponential integrals), 1029
elaborated type specifier, see class name, elaborated
elaborated-type-specifier, 152, 1270
element access functions, 902
either-group, 395, 1278
either-groups, 395, 1278
elision
   copy, see constructor, copy, elision
   copy constructor, see constructor, copy, elision
   move constructor, see constructor, move, elision
ellipsis
copy, see constructor, copy, elision
   overload resolution and, 287
eccentric integrals
   complete Π, 1027
   complete Ε, 1027
   incomplete Π, 1029
   incomplete Ε, 1029
   incomplete F, 1028
either-group, 395, 1278
EmplaceConstructible into X from args, 755
empty future object, 1251
empty shared_future object, 1253
eempty-declaration, 139, 1269
enclosing-namespace-specifier, 159, 1271
encoded character type, 1110
encoding
   multibyte, 21
   encoding-prefix, 17, 1261
end
unordered associative containers, 781
end-of-file, 539
\texttt{endif-line}, 395, 1278
engine, \textit{see} random number engine
engine adaptor, \textit{see} random number engine adaptor
engines with predefined parameters
\texttt{default_random_engine}, 967
\texttt{knuth_b}, 967
\texttt{minstd_rand}, 966
\texttt{minstd_rand0}, 966
\texttt{mt19937}, 966
\texttt{mt19937_64}, 967
\texttt{ranlux24}, 967
\texttt{ranlux24_base}, 967
\texttt{ranlux48}, 967
\texttt{ranlux48_base}, 967
entity, 24
templated, 307
\texttt{enum}, 61
overloading and, 277
type of, 156, 157
underlying type, \textit{see} type, underlying
\texttt{enum name}
\texttt{typedef}, 144
\texttt{enum-base}, 156, 1270
\texttt{enum-head}, 156, 1270
\texttt{enum-head-name}, 156, 1270
\texttt{enum-key}, 156, 1270
\texttt{enum-name}, 156, 1270
\texttt{enum-specifier}, 156, 1270
enumeration, 156, 157
linkage of, 46
scoped, 157
unscoped, 157
enumeration scope, 33
enumeration type
conversion to, 106
\texttt{static_cast}
conversion to, 106
enumerator
definition, 26
value of, 157
\texttt{enumerator}, 156, 1270
\texttt{enumerator-definition}, 156, 1270
\texttt{enumerator-list}, 156, 1270
environment
program, 71
epoch, 641
\texttt{equal_range}
unordered associative containers, 780
\texttt{equality-expression}, 121, 1267
equivalence
template type, 322
type, 143, 213
equivalent
expressions, 337
function templates, 337
functionally, \textit{see} functionally equivalent
\texttt{template-heads}, 337
\texttt{template-parameters}, 337
equivalent parameter declarations, 277
overloading and, 277
equivalent-key group, 772
equivalently-valued, 425
\texttt{Erasable} from $X$, 755
\texttt{erase}
unordered associative containers, 780
<\texttt{errno.h}>, 476
escape character, \textit{see} backslash character
escape sequence
undefined, 18
\texttt{escape-sequence}, 17, 1261
Eulerian integral of the first kind, \textit{see} beta
evaluation, 64
order of argument, 100
signal-safe, 471
unspecified order of, 65, 72
unspecified order of argument, 100
unspecified order of function call, 100
example
array, 186
class definition, 216
\texttt{const}, 183
constant pointer, 183
constructor, 249
destructor and initialization, 259
declaration, 25, 189
declarator, 181
definition, 25
\texttt{delete}, 258
derived class, 227
destructor and \texttt{delete}, 258
episodes, 187
enumeration, 158
explicit destructor call, 257
explicit qualification, 230
friend, 214
friend function, 242
function declaration, 188
function definition, 192
linkage consistency, 142
local class, 226
member function, 219, 242
nested class, 222
nested class definition, 223, 246
nested class forward declaration, 223
nested type name, 223
pointer-to-member, 185
pure virtual function, 236
scope of \texttt{delete}, 258
scope resolution operator, 230
static member, 221
subscripting, 186
type name, 181
\texttt{typedef}, 143
unnamed parameter, 193
variable parameter list, 187
virtual function, 234, 235
exception
arithmetic, 80
undefined arithmetic, 80
<exception>, 457, 1312
exception handling, 386–394
constructors and destructors, 388
currently handled exception, 390
exception object, 387, 388
constructor, 388
destructor, 388
function try block, 386
goto, 386
handler, 386, 387, 389–390, 433
active, 390
array in, 389
incomplete type in, 389
match, 389–390
pointer to function in, 389
rvalue reference in, 389
memory, 388
nearest handler, 387
rethrow, 124, 125, 388
rethrowing, 388
switch, 386
terminate called, 125, 387, 391
throwing, 124, 387
try block, 386
exception object, see exception handling, exception object
exception specification, 390–393
noexcept, 116
constant expression and, 391
non-throwing, 390
potentially-throwing, 390
virtual function and, 391
<exception_declaration>, 386, 1277
exclusive-or-expression, 122, 1267
<execution>, 655
execution agent, 1215
execution policy, 654
execution step, 69
exit, 71, 73, 136
explicit type conversion, see casting explicit-instantiation, 360, 1277
explicit-specialization, 362, 1277
explicitly captured, 92
explicitly initialized elements
aggregate, 200
exponent-part, 19, 1262
exponential integrals Ei, 1029
exponential_distribution
probability density function, 975
expr-or-braced-init-list, 196, 1274
expression, 80–129
additive operators, 118
alignof, 116
assignment and compound assignment, 125
bitwise AND, 122
bitwise exclusive OR, 122
bitwise inclusive OR, 122
cast, 101, 116–117
class member access, 101
comma, 125
conditional operator, 123
cost cast, 107
current, 126, 129
core constant, 126
decrement, 103, 110
delete, 115
dynamic cast, 103
equality operators, 121
equivalent, see equivalent, expressions
fold, 95–96
function call, 99
functionally equivalent, see functionally equivalent, expressions
increment, 102, 110
integral constant, 128
lambda, 87–95
left-shift-operator, 119
logical AND, 123
logical OR, 123
multiplicative operators, 118
new, 110
noexcept, 116
order of evaluation of, 7
parenthesized, 85
pointer-to-member, 117
pointer-to-member constant, 109
postfix, 99–108
potentially constant evaluated, 129
potentially evaluated, 26
primary, 84–99
pseudo-destructor call, 101
reference, 82
reinterpret cast, 106
relational operators, 120
requires, 96–99
right-shift-operator, 119
rvalue reference, 81
sizeof, 110
spaceship, 120
static cast, 105
three-way comparison, 120
throw, 124
type identification, 104
type-dependent, 347
unary, 108–116
unary operator, 109
value-dependent, 347
expression, 126, 1267
expression-list, 99, 1265
expression-statement, 131, 1268
extend, see namespace, extend
extended alignment, 57
extended integer type, 60
extended signed integer type, 60
extended unsigned integer type, 60
extern, 141
   linkage of, 142
extern "C", 418, 428
extern "C++", 418, 428
external linkage, 46
extreme_value_distribution
   probability density function, 978
F (incomplete elliptic integrals), 1028
facet, 712
fallback-separator, 1117
file, 1109
file attributes, 1135
   cached, 1135
file system, 1109
file system race, 1110
file, source, see source file
filename, 1117
filename, 1117
filesystem>, 1110
final overrider, 233
find
   unordered associative containers, 780
finite state machine, 1158
fisher_f_distribution
   probability density function, 981
floating literal, see literal, floating
floating-literal, 18, 1262
floating-point literal, see literal, floating
floating-point type, 60
   implementation-defined, 60
floating-suffix, 19, 1262
fold
   binary, 95
   unary, 95
fold-expression, 95, 1264
fold-operator, 95, 1264
for
   scope of declaration in, 135
for-range-declaration, 133, 1268
for-range-initializer, 133, 1268
format specifier, 1158
forward, 490
forward progress guarantees
   concurrent, 69
   delegation of, 70
   parallel, 70
   weakly parallel, 70
forward_list>, 784
forwarding call wrapper, 596
forwarding reference, 374
fractional-constant, 19, 1262
free store, see also new, delete, 257
freestanding implementation, 6
friend
   virtual and, 235
   access specifier and, 244
   class access and, 242
   inheritance and, 244
   local class and, 244
   template and, 330
friend function
   access and, 242
   inline, 244
   linkage of, 243
   member function and, 242
   nested class, 223
fstream>, 1091
full-expression, 64
function, see also friend function; member
   function; inline function; virtual function
   allocation, 55, 112
   comparison, 408
   conversion, 254
   deallocation, 55, 258
   definition, 26
   global, 428, 431
   handler, 409
   handler of type, 389
   linkage specification overloaded, 172
   modifier, 409
   named by an expression, 26
   needed for constant evaluation, 129
   observer, 409
   operator, 299
   overload resolution and, 280
   overloading and pointer versus, 277
   parameter of type, 187
   pointer to member, 118
   program semantics affected by the existence
      of a function definition, 357
   replacement, 409
   reserved, 410
   template parameter of type, 309
   viable, 279
   virtual function call, 100
   virtual member, 429
function argument, see argument
function call, 101
   recursive, 101
   undefined, 107
function call operator
   overloaded, 301
function object, 593
   overloaded, 301
   binders, 605–607
   mem_fn, 607
   reference_wrapper, 597
   type, 593
   wrapper, 607–611
function parameter, see parameter
function parameter pack, 328
function parameter scope, 31
function pointer type, 61
function return, see return
function return type, see return type
Index 1361

function try block, see exception handling, function try block
function, overloaded, see overloading
function, virtual, see virtual function
function-body, 192, 1273
function-definition, 192, 1273
function-like macro, see macro, function-like
function-specifier, 143, 1269
function-try-block, 386, 1277
<functional>, 594, 1316
functionally equivalent
expressions, 337
function templates, 338
template-heads, 337
functions
candidate, 354
fundamental alignment, 57
fundamental type
destructor and, 257
fundamental type conversion, see conversion, user-defined
future
shared state, 1247
<future>, 1245
gamma_distribution
probability density function, 976
generate
seed sequence requirement, 952
generated destructor, see destructor, default
generation algorithm
discard_block_engine, 964
independent_bits_engine, 965
linear_congruential_engine, 960
mersenne_twister_engine, 961
shuffle_order_engine, 966
subtract_with_carry_engine, 962
generic lambda, 87
geometric_distribution
discrete probability function, 973
global, 32
global namespace, 32
global namespace scope, 32
global scope, 32
glvalue, 80
goto
and handler, 386
and try block, 386
initialization and, 137
grammar, 1259
regular expression, 1195
grep, 1167
group, 395, 1277
group-part, 395, 1277
H_n (Hermite polynomials), 1029
h-char, 12, 1260
h-char-sequence, 12, 1260
h-pp-tokens, 396, 1278
h-preprocessing-token, 396, 1278
handler, see exception handling, handler
handler, 386, 1277
handler-seq, 386, 1277
happens after, 67
happens before, 67
hard link, 1109
has-include-expression, 396, 1278
hash
instantiation restrictions, 614
hash code, 772
hash function, 771
hash tables, see unordered associative containers
hash_function
unordered associative containers, 775
hasher
unordered associative containers, 773
header
C, 428, 431, 1304
C library, 418
C++ library, 416
name, 12–13
header-name, 12, 1259
headers
C library, 1304
Hermite polynomials H_n, 1029
hex-quad, 10, 1259
hexadecimal-digit, 16, 1261
hexadecimal-digit-sequence, 16, 1261
hexadecimal-escape-sequence, 17, 1262
hexadecimal-floating-literal, 18, 1262
hexadecimal-fractional-constant, 19, 1262
hexadecimal-literal, 15, 1261
hexadecimal-prefix, 15, 1261
hiding, see name hiding
high-order bit, 48
hosted implementation, 6
I_n (Bessel functions), 1027
id
qualified, 86
id-expression, 85
id-expression, 85, 1263
identical
atomic constraints, see atomic constraint, identical
identifier, 13–14, 86, 140
identifier, 13, 1260
identifier label, 131
identifier-list, 396, 1278
identifier-nondigit, 13, 1260
if-group, 395, 1277
if-section, 395, 1277
ill-formed program, see program, ill-formed
immediate subexpression, 63
implementation
freestanding, 417
hosted, 417
implementation limits, see limits, implementation
implementation-defined behavior, see behavior, implementation-defined
implementation-dependent, 1064
implementation-generated, 25
implicit conversion, see conversion, implicit
implicit object parameter, 280
implicitly-declared default constructor, see constructor, default, 248
implied object argument, 280
implicit conversion sequences, 280
non-static member function and, 280
inclusion
conditional, see preprocessing directive, conditional inclusion
source file, see preprocessing directives, source-file inclusion
inclusive-or-expression, 122, 1267
incomplete, 119
increment operator
overloaded, see overloading, increment operator
independent_bits_engine
generation algorithm, 965
state, 964
textual representation, 965
transition algorithm, 965
indeterminate value, 197
indeterminately sequenced, 65
indirection, 109
inheritance, 227
using-declaration and, 163
init-capture, 91, 1264
init-declarator, 179, 1272
init-declarator-list, 179, 1272
init-statement, 130, 1267
initialization, 71, 196–210
aggregate, 199
array, 199
array of class objects, 203, 259
automatic, 137
automatic object, 196
base class, 260, 261
by inherited constructor, 264
class member, 198
class object, see also constructor, 199,
259–265
cast, 148, 199
cast member, 261
cast member, 261
constant, 71
constructor and, 259
copy, 198
default, 196
default constructor and, 259
definition and, 140
direct, 198
dynamic, 71
dynamic block-scope, 137
dynamic non-local, 72
explicit, 259
jump past, 137
list-initialization, 206–210
local static, 137
local thread local, 137
member, 260
member function call during, 263
member object, 261
non-vacuous, 50
order of, 72, 228
order of base class, 262
order of member, 262
order of virtual base class, 262
overloaded assignment and, 259
parameter, 100
reference, 184, 204
reference member, 261
runtime, 71
static and thread, 71
static member, 221
static object, 71, 196
union, 203, 225
virtual base class, 270
zero-initialization, 196
initializer
base class, 193
member, 193
pack expansion, 264
scope of member, 263
temporary and declarator, 250
initializer, 196, 1273
initializer-clause, 196, 1273
initializer-list, 196, 1273
initializer-list constructor, 207
seed sequence requirement, 952
<initializer_list>, 461
initializing declaration, 199
injected-class-name, 211
inline, 431
inline
linkage of, 46
inline function, 147
insert
unordered associative containers, 776, 777
instantiation
explicit, 360
point of, 353	template implicit, 356
instantiation units, 10
integer literal, see literal, integer
integer representation, 56
integer type, 60
integer-literal, 15, 1260
integer-suffix, 16, 261
integral type, 60
implementation-defined sizeof, 60
inter-thread happens before, 67
internal linkage, 46
interval boundaries
Index
piecewise_constant_distribution, 984
piecewise_linear_distribution, 985
<inttypes.h>, 1157
invalid pointer value, 61
invocation
  macro, 400
<iomanip>, 1060
<ios>, 1037
<iosfwd>, 1033
<iostream>, 1035
isctype
  regular expression traits, 1160
<iso646.h>, 1301
<istream>, 1059
iteration-statement, 133, 136, 1268
<iterator>, 860, 1320
jn (spherical Bessel functions), 1030
Jν (Bessel functions), 1028
Jessie, 253
jump-statement, 136, 1268
K (complete elliptic integrals), 1027
Kν (Bessel functions), 1028
key_eq
  unordered associative containers, 775
key_equal
  unordered associative containers, 773
key_type
  unordered associative containers, 772
keyword, 14, 1259
Ln (Laguerre polynomials), 1030
Lm (associated Laguerre polynomials), 1026
label, 136
  case, 131–133
  default, 131–133
  scope of, 31, 131
labeled-statement, 130, 1268
Laguerre polynomials
  Ln, 1030
  Lm, 1026
lambda-capture, 91, 1264
lambda-declarator, 87, 1264
lambda-expression, 87, 1264
lambda-introducer, 87, 150, 1264
lattice, see DAG, subobject
layout
  bit-field, 222
  class object, 217, 228
layout-compatible type, 59
left shift
  undefined, 119
left shift operator, see operator, left shift
Legendre functions Ymν, 1031
Legendre polynomials
Pν, 1030
Pmν, 1026
lexical conventions, see conventions, lexical
LIA-1, 1328
library
  C standard, 408, 414, 416, 418, 1301, 1304
  C++ standard, 407, 429, 430, 432, 433
library clauses, 8
lifetime, 50
limits
  implementation, 4
<limits>, 438
line splicing, 9
linear_congruential_engine
  generation algorithm, 960
  modulus, 961
  state, 960
  textual representation, 961
  transition algorithm, 960
link, 1109
linkage, 24, 46–48
  const and, 46
  external, 46, 418, 428
  implementation-defined object, 173
  inline and, 46
  internal, 46
  no, 46, 47
  static and, 46
linkage specification, see specification, linkage
linkage-specification, 171, 1271
<list>, 784
list-initialization, 206
literal, 15–23, 84
  base of integer, 16
  binary, 16
  boolean, 21
  char16_t, 17
  char32_t, 17
  character, 17
  char16_t, 17
  char32_t, 17
  ordinary, 17
  UTF-8, 17
  wide, 18
constant, 15
decimal, 16
decimal floating, 19
double, 19
float, 19
floating, 18, 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
pointer, 21
string, 19, 20
char16_t, 20
char32_t, 20
narrow, 20
raw, 11, 20
type of, 20
UTF-8, 20
wide, 20
type of character, 17
type of floating-point, 19
type of integer, 16
unsigned, 16
user-defined, 22
literal, 15, 1260
literal type, 59
literal-operator-id, 302, 1275
living dead
name of, 427
load_factor
unordered associative containers, 781
local class, 226
friend, 244
member function in, 218
scope of, 226
local entity, 24
local lambda expression, 91
locale, 1158–1160, 1167
locale-specific behavior, see behavior
locale-specific <locale>, 708, 709
locale-specific behavior, see behavior
locale-specific <locale.h>, 749
lock-free execution, 69
logical-and-expression, 123, 1267
logical-or-expression, 123, 1267
lognormal_distribution
probability density function, 979
long
typedef and, 141
long-long-suffix, 16, 1261
long-suffix, 16, 1261
lookup
argument-dependent, 38
class member, 40, 45
elaborated type specifier, 44–45
member name, 229
name, 24, 34–46
namespace aliases and, 46
namespace member, 41
qualified name, 39–44
template name, 341
unqualified name, 34
using-directives and, 46
lookup_classname
regular expression traits, 1160, 1197
lookup_collatename
regular expression traits, 1160
low-order bit, 48
lowercase, 414
lparen, 395, 1278
lvalue, 80, 1283
lvalue reference, 61, 184
Lvalue-Callable, 609
macro
argument substitution, 400
function-like, 399
arguments, 400
masking, 431
name, 399
object-like, 399
pragma operator, 406
predefined, 404
replacement, 399–403
replacement list, 399
rescanning and replacement, 401
scope of definition, 402
main function, 70–71
implementation-defined linkage of, 71
implementation-defined parameters to, 71
parameters to, 71
return from, 71, 73
make progress
thread, 69
<map>, 809
match_results
as sequence, 1181
matched, 1158
mathematical special functions, 1026–1031
max
random number distribution requirement, 956
uniform random bit generator requirement, 952
max_bucket_count
unordered associative containers, 780
max_load_factor
unordered associative containers, 781
mean
normal_distribution, 978
poisson_distribution, 975
mem-initializer, 260, 1275
mem-initializer-id, 260, 1275
mem-initializer-list, 260, 1275
member
class static, 54
default initializer, 216
enumerator, 158
static, 215, 220
template and static, 325
member access operator
overloaded, 301
member data
  static, 221
member function, 215
  call undefined, 219
  class, 218
  const, 219
  const volatile, 219
  constructor and, 249
  destructor and, 256
  friend, 243
  inline, 218
  local class, 226
  nested class, 246
  non-static, 219
  overload resolution and, 280
  static, 215, 221
this, 220
union, 224
volatile, 219
member names, 32
member of an unknown specialization, 349
member of the current instantiation, see current instantiation, member of the
member pointer to, see pointer to member
member subobject, 49
member-declaration, 215, 1274
member-declarator, 215, 1274
member-declarator-list, 215, 1274
member-specification, 214, 1274
members, 32
<memory>, 539, 1318, 1319
memory location, 48
memory management, see also new, delete
memory model, 48–49
<memory_resource>, 579
mersenne_twister_engine
  generation algorithm, 961
  state, 961
  textual representation, 962
  transition algorithm, 961
message
  diagnostic, 3, 6
min
  random number distribution requirement, 956
  uniform random bit generator requirement, 952
modifiable, 82
modification order, 66
more constrained, 322
more cv-qualified, 62
more specialized, 334, 377
  class template, 334
  function template, 377
most derived class, 50
most derived object, 50
  bit-field, 50
  zero size subobject, 50
move
  class object, see constructor, move, see
  assignment operator, move
move, 490
MoveInsertable into X, 754
multi-pass guarantee, 858
multibyte character, see character, multibyte
multicharacter literal, see literal, multicharacter
multiline, 1167
multiple inheritance, 227, 228
virtual and, 235
multiple threads, see threads, multiple
  multiplicative-expression, 118, 1266
mutable, 141
mutable iterator, 855
<mutex>, 1221
mutex types, 1222
n_n (spherical Neumann functions), 1031
N_ν (Neumann functions), 1028
name, 13, 24, 85
  address of cv-qualified, 109
  dependent, 347, 353
  elaborated
    enum, 152
  global, 32
  length of, 13
  macro, see macro, name
  point of declaration, 30
  predefined macro, see macro, predefined
  qualified, 39
  reserved, 427
  same, 24
  scope of, 29
  unqualified, 34
  zombie, 427
name class, see class name
name hiding, 30, 34, 86, 137
  class definition, 213
  function, 278
  overloading versus, 278
  user-defined conversion and, 252
  using-declaration and, 166
name space
  label, 131
  named by an expression, 26
named-namespace-definition, 159, 1271
namespace, 415, 1304
  alias, 162
  associated, 38
  definition, 159
  extend, 159
  global, 14
  member definition, 161
  unnamed, 161
namespace-alias, 162, 1271
namespace-alias-definition, 162, 1271
namespace-body, 159, 1271
namespace-definition, 159, 1271
namespace-name, 159, 1271
namespaces, 159–171
NaN, 1026
narrowing conversion, 210
native encoding, 1119
native pathname format, 1114
NDEBUG, 417
needed
   exception specification, 392
needed for constant evaluation, 129
negative_binomial_distribution
   discrete probability function, 974
nested class, 222
   local class, 226
   scope of, 222
nested within, 50
nested-name-specifier, 86, 1264
nested-namespace-definition, 159, 1271
nested-requirement, 98, 1265
Neumann functions
   \( N \), 1028
   \( n \), 1031
<new>, 448
new, 54, 110, 111
   array of class objects and, 114
   constructor and, 114
   default constructor and, 114
   exception and, 114
   initialization and, 114
   operator
      replaceable, 429
      scoping and, 111
      storage allocation, 110
      type of, 257
      unspecified constructor and, 114
   unspecified order of evaluation, 114
new-declarator, 111, 1266
new-expression, 110, 1266
   placement, 113
new-extended alignment, 57
new-initializer, 111, 1266
new-line, 396, 1278
new-placement, 111, 1266
new-type-id, 111, 1266
new_handler, 55
   no linkage, 46
nodeclspec-function-declaration, 139, 1269
noexcept, 116
noexcept-expression, 116, 1266
noexcept-specifier, 390, 1277
non-initialization odr-use, see odr-use,
   non-initialization
non-static data member, 215
non-static member, 215
non-static member function, 215
non-throwing exception specification, 390
nondigit, 13, 1260
nonzero-digit, 15, 1261
noptr-abstract-declarator, 180, 1273
noptr-abstract-pack-declarator, 181, 1273
noptr-declarator, 180, 1272
noptr-new-declarator, 111, 1266
normal distributions, 978–982
normal form
   constraint, 321
   path, 1118
normal_distribution
   mean, 978
   probability density function, 978
   standard deviation, 978
normalization
   constraint, see constraint, normalization
   path, see path, normalization
normative references, see references, normative notation
   syntax, 8
NTBS, 414, 1094, 1310, 1311
   static, 414
NTCTS, 409
NTMBS, 414
   static, 414
null character, see character, null
null pointer value, 61
null statement, 131
null wide character, see wide-character, null
   number
      hex, 18
      octal, 18
      preprocessing, 13
      subnormal, 439, 440, 442, 443
<numeric>, 1005
numeric_limits, 438
   specializations for arithmetic types, 60
object, see also object model, 24, 49
   byte copying and, 57–58
   complete, 49
   const, 62
   const volatile, 62
   definition, 26
   destructor and placement of, 257
   destructor static, 73
   linkage specification, 173
   local static, 54
   nested within, 50
   providing storage for, 49
   unnamed, 249
   volatile, 62
object class, see also class object
object expression, 102, 118
object lifetime, 50–53
object model, 49–50
object pointer type, 61
object representation, 58
object temporary, see temporary
object type, 59
   incompletely-defined, 58
object, exception, see exception handling,
   exception object
object-like macro, see macro, object-like
observable behavior, see behavior, observable
octal-digit, 15, 1261
octal-escape-sequence, 17, 1262
octal-literal, 15, 1261
odr-usable, 27
odr-use
    non-initialization, 72
odr-used, 27
one-definition rule, 26–29
opaque-enum-declaration, 156, 1270
operating system dependent, 1109
operation
    atomic, 65–70
operator, 15, 300
    **, 125
    +=, 110, 125
    -=, 125
    /=, 125
    >>=, 125
    %*, 125
    &*, 125
    !*, 125
    |*, 125
    !, 125
    addition, 118
    additive, 118
    address-of, 108
    assignment, 125, 414
    bitwise, 122
    bitwise AND, 122
    bitwise exclusive OR, 122
    bitwise inclusive OR, 122
    cast, 108, 180
    class member access, 101
    comma, 125
    conditional expression, 123
    copy assignment, see assignment operator, copy
decrement, 103, 108, 110
division, 118
equality, 121
defaulted, 275
function call, 99, 299
greater than, 120
greater than or equal to, 120
increment, 102, 108, 110
indirection, 108
inequality, 121
defaulted, 275
left shift, 119
less than, 120
less than or equal to, 120
logical AND, 123
logical negation, 108, 109
logical OR, 123
move assignment, see assignment operator, move
multiplication, 118
multiplicative, 118
ones’ complement, 108, 109
overloaded, 80, 299
pointer to member, 117
pragma, see macro, pragma operator
precedence of, 7
relational, 120
defaulted, 275
remainder, 118
right shift, 119
scope resolution, 40, 86, 112, 218, 227, 236
side effects and comma, 126
side effects and logical AND, 123
side effects and logical OR, 123
sizeof, 108, 110
spaceship, 120
subscripting, 99, 299
subtraction, 118
tree-way comparison, 120
defaulted, 274
unary, 108, 109
unary minus, 108, 109
unary plus, 108, 109
operator, 299, 1275
operator delete, see also delete, 112, 116, 258
operator new, see also new, 112
operator overloading, see overloading, operator
operator use
    scope resolution, 221
operator!=
    random number distribution requirement, 956
    random number engine requirement, 954
operator()
    random number distribution requirement, 956
    random number engine requirement, 953
    uniform random bit generator requirement, 952
operator-function-id, 299, 1275
operator<<
    random number distribution requirement, 957
    random number engine requirement, 954
operator==
    random number distribution requirement, 956
    random number engine requirement, 954
operator>>
    random number distribution requirement, 957
    random number engine requirement, 954
operators
    built-in, 80
optimization of temporary, see temporary, elimination of
<optional>, 505
optional object, 505
order of evaluation in expression, see expression, order of evaluation of
order of execution
    base class constructor, 249
    base class destructor, 256
    constructor and static objects, 260
constructor and array, 259
destructor, 256
destructor and array, 256
member constructor, 249
member destructor, 256
ordering
  function template partial, see template, function, partial ordering
ordinary character literal, 17
ordinary string literal, 20
<ostream>, 1060
over-aligned type, see type, over-aligned
overflow, 80
  undefined, 80
overloaded function, see overloading
  address of, 109, 298
overloaded operator, see overloading, operator
  inheritance of, 300
overloading, 188, 213, 276–305, 336
  access control and, 279
  address of overloaded function, 298
  argument lists, 279–287
  array versus pointer, 277
  assignment operator, 300
  binary operator, 300
  built-in operators and, 303
  candidate functions, 279–287
  declaration matching, 278
  declarations, 276
  example of, 276
  function call operator, 301
  function versus pointer, 277
  member access operator, 301
  operator, 299–303
  prohibited, 276
resolution, 279–298
  best viable function, 287–300
  contexts, 279
  function call syntax, 281–282
  function template, 384
  implicit conversions and, 290–298
  initialization, 284–286
  operators, 282
  scoping ambiguity, 230
  template, 338
  template name, 341
  viable functions, 287–300
  subscripting operator, 301
  unary operator, 300
  user-defined literal, 302
  using directive and, 170
  using-declaration and, 167
overloads
  floating-point, 949
overrider
  final, 233
own, 552
\(P^m_\ell\) (associated Legendre polynomials), 1026
\(P^m_\ell\) (complete elliptic integrals), 1027
\(P^m_\ell\) (incomplete elliptic integrals), 1029
piecewise construction, 493
piecewise_constant_distribution
  interval boundaries, 984
  probability density function, 984
  weights, 984
piecewise_linear_distribution
  interval boundaries, 985
  probability density function, 985
  weights at boundaries, 985
placeholder type deduction, 154
placement new-expression, 113

placement syntax
new, 113
plain lock-free atomic operation, 470
pm-expression, 117, 1266
POD, 1320
point, 61
point of declaration, 30
pointer, see also void*
    composite pointer type, 82
    integer representation of safely-derived, 56
    safely-derived, 56
    to traceable object, 56, 433
    zero, 78
pointer literal, see literal, pointer
pointer past the end of, 61
pointer to, 61
pointer to member, 61, 117
pointer-interconvertible, 61
pointer-literal, 21, 1263
Poisson distributions, 975–978
poisson_distribution
    discrete probability function, 975
    mean, 975
pool resource classes, 584
pools, 584
population, 920
POSIX, 2
    extended regular expressions, 1167
    regular expressions, 1167
postfix ++ and --
    overloading, 301
postfix ++, 102
postfix --, 103
postfix-expression, 99, 1265
potential results, 26
potential scope, 29
potentially concurrent, 68
potentially constant evaluated, 129
potentially evaluated, 26
potentially-throwing
    exception specification, 390
    expression, 391
pp-number, 13, 1260
pp-tokens, 396, 1278
precedence of operator, see operator, precedence of
preferred-separator, 1117
prefix
    L, 18, 20
    R, 20
    U, 17, 20
    u, 17, 20
    u8, 17, 20
prefix ++ and --
    overloading, 301
prefix ++, 110
prefix --, 110
preprocessing directives, 395–406
error, 404
header inclusion, 398
line control, 404
macro replacement, see macro, replacement
null, 404
pragma, 404
source-file inclusion, 398
preprocessing-file, 395, 1277
preprocessing-op-or-punc, 15, 1260
preprocessing-token, 11, 1259
primary class template, see template, primary
primary equivalence class, 1158
primary-expression, 84, 1263
private, see access control, private
probability density function
cauhcy_distribution, 980
chi_squared_distribution, 980
exponential_distribution, 975
extreme_value_distribution, 978
fisher_f_distribution, 981
gamma_distribution, 976
lognormal_distribution, 979
normal_distribution, 978
piecewise_constant_distribution, 984
piecewise_linear_distribution, 985
student_t_distribution, 982
uniform_real_distribution, 971
weibull_distribution, 977
program, 46
    ill-formed, 4
    start, 70–73
    termination, 73–74
    well-formed, 5, 7
program execution, 6–65
    abstract machine, 6
    as-if rule, see as-if rule
program semantics
    affected by the existence of a variable or
    function definition, 357
promoted arithmetic type, 303
promoted integral type, 303
promotion
    bool to int, 78
    default argument promotion, 101
    floating-point, 78
    integral, 77
protected, see access control, protected
protection, see access control, 432
prototype parameter
    concept, 341
provide storage, 49
prvalue, 80
pseudo-destructor-name, 101
pseudo-destructor-name, 99, 1265
ptr-abstract-declarator, 180, 1265
ptr-declarator, 180, 1272
ptr-operator, 180, 1272
ptrdiff_t, 119
implementation-defined type of, 119

public, see access control, public

punctuator, 15
pure-specifier, 215, 1274

q-char, 12, 1260
q-char-sequence, 12, 1260
qualification
explicit, 39
qualified-concept-name, 308, 1276
qualified-id, 86, 1264
qualified-namespace-specifier, 162, 1271
<queue>, 847

r-char, 20, 1263
r-char-sequence, 19, 1262
<brandom>, 957

random number distribution
bernoulli_distribution, 972
binomial_distribution, 973
cauchy_distribution, 980
chi_squared_distribution, 980
discrete_distribution, 983
exponential_distribution, 975
extreme_value_distribution, 978
fisher_f_distribution, 981
gamma_distribution, 976
gaussian_distribution, 973
lognormal_distribution, 979
negative_binomial_distribution, 974
normal_distribution, 978
piecewise_constant_distribution, 984
piecewise_linear_distribution, 985
poisson_distribution, 975
requirements, 955–957
student_t_distribution, 982
uniform_int_distribution, 970
uniform_real_distribution, 971
weibull_distribution, 977
random number distributions
Bernoulli, 972–975
normal, 978–982
Poisson, 975–978
sampling, 983–987
uniform, 970–972
random number engine
linear_congruential_engine, 960
mersenne_twister_engine, 961
requirements, 952–954
subtract_with_carry_engine, 962
with predefined parameters, 966–967
random number engine adaptor
discard_block_engine, 964
independent_bits_engine, 964
shuffle_order_engine, 965
with predefined parameters, 966–967
random number generation, 950–987
distributions, 970–987
engines, 960–966

predefined engines and adaptors, 966–967
requirements, 951–957
synopsis, 957–959
utilities, 968–970

random number generator, see uniform random
bit generator

random_device
implementation leeway, 967
<bratio>, 636
raw string literal, 20
raw-string, 19, 1262
ready, 1181, 1248
redefinition
typedef, 143
ref-qualifier, 180, 1272
reference, 61
assignment to, 125
call by, 101
forwarding, 373
lvalue, 61
null, 184
rvalue, 61
sizeof, 110
reference collapsing, 184
reference-compatible, 204
reference-related, 204
references
normative, 2<brregex>, 1160
regex_iterator
end-of-sequence, 1190
regex_token_iterator
end-of-sequence, 1192
regex_traits
specializations, 1169
region
declarative, 24, 29
intervening, 30
register storage class, 1297
regular expression, 1158–1197
grammar, 1195
matched, 1158
requirements, 1159
regular expression traits, 1195
char_class_type, 1159
isctype, 1160
lookup_classname, 1160, 1197
lookup_collatename, 1160
requirements, 1159, 1169
transform, 1159, 1196
transform_primary, 1160, 1197
translate, 1159, 1196
translate_nocase, 1159, 1196
rehash
unordered associative containers, 781
reinterpret_cast, see cast, reinterpret
relational-expression, 120, 1267
relative path, 1114
relative-path, 1117
relaxed pointer safety, 56
release sequence, 66
remainder operator, see operator, remainder replacement
    macro, see macro, replacement
replacement-list, 396, 1278
representation
    object, 58
    value, 58
represents the address, 61
requirement, 97
    compound, 97
    nested, 98
    simple, 97
    type, 97
requirement, 96, 1265
requirement-body, 96, 1265
requirement-parameter-list, 96, 1265
requirement-seq, 96, 1265
requirements, 411
    Allocator, 422
    container, 750, 772, 785, 786, 1181
    not required for unordered associated containers, 771
CopyAssignable, 418
CopyConstructible, 418
DefaultConstructible, 418
Destructible, 418
EqualityComparable, 418
Hash, 421
iterator, 855
LessThanComparable, 418
MoveAssignable, 418
MoveConstructible, 418
NullablePointer, 421
numeric type, 940
random number distribution, 955–957
random number engine, 952–954
regular expression traits, 1159, 1169
seed sequence, 951–952
sequence, 1181
uniform random bit generator, 952
unordered associative container, 772
requires-clause, 306, 1276
requires-expression, 96, 1265
rescanning and replacement, see macro, rescanning and replacement
reserved identifier, 13
reset, 553
    random number distribution requirement, 956
resolution, see overloading, resolution
restriction, 430, 431, 433
    address of bit-field, 222
    anonymous union, 225
    bit-field, 222
    constructor, 248, 249
    destructor, 255
    extern, 142
    local class, 226
    operator overloading, 299
    overloading, 300
    pointer to bit-field, 222
    reference, 184
    static, 142
    static member local class, 226
    union, 224
result
    glvalue, 81
    prvalue, 81
result object, 81
result_type
    entity characterization based on, 950
    random number distribution requirement, 956
    seed sequence requirement, 951
    uniform random bit generator requirement, 952
rethrow, see exception handling, rethrow
return, 135, 136
    and handler, 386
    and try block, 386
    constructor and, 136
    reference and, 204
return statement, see return
return type, 188
    covariant, 234
    overloading and, 276
return-type-requirement, 97, 1265
right shift
    implementation-defined, 119
    right shift operator, see operator, right shift
root-directory, 1117
root-name, 1117
rounding, 78
rvalue, 80
    lvalue conversion to, see conversion,
    lvalue-to-rvalue, 1283
rvalue reference, 61, 184
s-char, 19, 1262
s-char-sequence, 19, 1262
safely-derived pointer, 56
    integer representation, 56
sample, 920
sampling distributions, 983–987
scalar type, 59
scope, 1, 24, 29–34, 140
    anonymous union at namespace, 225
    block, 31
    class, 32
    declarations and, 29–31
    destructor and exit from, 136
    enumeration, 33
    exception declaration, 31
    function, 31
    function parameter, 31
    function prototype, see scope, function parameter
    Index
global, 32
global namespace, 32
iteration-statement, 133
macro definition, see macro, scope of
definition
name lookup and, 34–46
namespace, 32
overloading and, 278
potential, 29
selection-statement, 131
template parameter, 33
scope name hiding and, 34
scope resolution operator, see operator, scope
resolution
<scoped_allocator>, 588
seed
random number engine requirement, 953
seed sequence, 951
requirements, 951–952
selection-statement, 131, 1268
semantics
class member, 101
separate compilation, see compilation, separate
separate translation, see compilation, separate
sequence
ambiguous conversion, 291
implicit conversion, 290
standard conversion, 75
sequence constructor
seed sequence requirement, 951
sequenced after, 65
sequenced before, 65
sequencing operator, see operator, comma
<set>, 810
<setjmp.h>, 470
setlocale, 414
shared lock, 1226
shared mutex types, 1226
shared state, see future, shared state
shared timed mutex type, 1227
<shared_mutex>, 1221
shift operator
left, see operator, left shift
right, see operator, right shift
shift-expression, 119, 1266
short
typedef and, 141
shuffle_order_engine
generation algorithm, 966
state, 965
textual representation, 966
transition algorithm, 965
side effects, 6, 64–68, 123, 131, 250, 262, 272, 402,
432
visible, 67, 68
sign, 19, 1262
signal, 65
signal-safe
.abort, 447
evaluation, see evaluation, signal-safe
.forward, 490
initializer_list functions, 461
memcpy, 705
memmove, 705
move, 490
move_if_noexcept, 490
numeric_limits members, 440
quick_exit, 448
signal, 471
type traits, 614
<signal.h>, 470
signature, 4, 5
signed
typedef and, 141
signed integer representation
ones’ complement, 60, 109, 157
signed magnitude, 60, 157
two’s complement, 60, 78, 157, 637, 1208
signed integer type, 60
significand, 19
similar types, 77
simple call wrapper, 596
simple-capture, 91, 1264
simple-declaration, 139, 1269
simple-escape-sequence, 17, 1262
simple-requirement, 97, 1265
simple-template-id, 312, 1276
simple-type-specifier, 150, 1270
size
seed sequence requirement, 952
size_t, 110
smart pointers, 561–575
source file, 9, 417, 429
source file character, see character, source file
space
white, 11
special member function, see constructor, see
destructor, see assignment operator
specialization
class template, 313
class template partial, 332
template, 354
template explicit, 362
specification
linkage, 171–173
extern, 171
implementation-defined, 171
nesting, 171
template argument, 367
specifications
C standard library exception, 433
C++, 433
specifier, 140–156
constexpr, 145
destructor, 145, 146
function, 145
cv-qualifier, 148
declaration, 140
explicit, 143
friend, 145, 432
function, 143
inline, 147
static, 141
storage class, 141
type, see type specifier
typedef, 143
virtual, 143
specifier access, see access specifier
spherical harmonics \( Y_{\ell}^{m} \), 1031

<sstream>, 1082
stable algorithm, 410, 432
<stack>, 848
stack unwinding, 388
standard
structure of, 8
standard deviation
normal_distribution, 978
standard integer type, 60
standard signed integer type, 59
standard unsigned integer type, 60
standard-layout class, 212
standard-layout struct, 212
standard-layout types, 59
standard-layout union, 212
start
program, 72
startup
program, 418, 429
state, 530
discard_block_engine, 964
independent_bits_engine, 964
linear_congruential_engine, 960
mersenne_twister_engine, 961
shuffle_order_engine, 965
subtract_with_carry_engine, 962
statement, 130–138
continue in for, 134
break, 135, 136
compound, 131
continue, 135, 136
declaration, 137
declaration in for, 135
declaration in if, 130
declaration in switch, 130, 133
declaration in while, 134
do, 133, 134
empty, 131
expression, 131
fallthrough, 177
for, 133, 134
goto, 131, 135, 136
if, 131
iteration, 133–135
jump, 135
labeled, 130
null, 131
range based for, 135
selection, 131–133
switch, 131, 132, 136
while, 133, 134
statement, 130, 1267
statement-seq, 131, 1268
static, 141
destruction of local, 137
linkage of, 46, 142
overloading and, 276
static data member, 215
static initialization, 71
static member, 215
static member function, 215
static storage duration, 54
static type, see type, static
static_assert, 140
static_assert not macro, 475
static_assert-declaration, 139, 1269
static_cast, see cast, static
<stdalign.h>, 1301, 1303
<stdarg.h>, 470
<stdatomic.h>, 416, 1301
<stdbool.h>, 1301, 1303
<stddata.h>, 18, 20
<stdexcept>, 472
<stdio.h>, 1156
<stdiolib.h>, 1304
<string.h>, 416, 1301
storage class, 24
storage duration, 53–57
automatic, 53, 54
class member, 57
dynamic, 53–56, 111
local object, 54
static, 53, 54
thread, 53, 54
storage management, see new, delete
storage-class-specifier, 141, 1269
stream
arbitrary-positional, 408
repositional, 410
<stdiostream>, 1052
strict pointer safety, 56
string
distinct, 21
null-terminated byte, 414
null-terminated character type, 409
null-terminated multibyte, 414
sizeof, 21
type of, 20
<string>, 665
string literal, see literal, string
string-literal, 19, 1262
<string.h>, 705
<string_view>, 695
stringize, see # operator
strongly happens before, 67

Index

1373
<strstream>, 1305
struct
    standard-layout, 212
struct
class versus, 211
structure, 211
structure tag, see class name
structured binding, 195
structured binding declaration, 140, 195
student_t_distribution
    probability density function, 982
sub-expression, 1159
subexpression, 64
subnormal number, see number, subnormal
subobject, see also object model, 49
subscripting operator
    overloaded, 301
subsequence rule
    overloading, 296
substitutability, 463
subsume, see constraint, subsumption
subtract_with_carry_engine
    carry, 962
generation algorithm, 962
state, 962
textual representation, 963
transition algorithm, 962
subtraction
    implementation-defined pointer, 119
subtraction operator, see operator, subtraction
suffix
    E, 19
e, 19
F, 19
f, 19
L, 16, 19
l, 16, 19
P, 19
p, 19
U, 16
u, 16
summary
    compatibility with ISO C, 1281
    compatibility with ISO C++ 2003, 1289
    compatibility with ISO C++ 2011, 1295
    compatibility with ISO C++ 2014, 1296
    compatibility with ISO C++ 2017, 1300
syntax, 1259
swappable, 420
swappable with, 420
switch
    and handler, 386
    and try block, 386
symbolic link, 1109
synchronize with, 66
<syncstream>, 1104
synonym, 162
type name as, 143
syntax
thread of execution, 65
thread storage duration, 54
thread_local, 141
threads
  multiple, 65–70
<threads.h>, 416, 1301
throw, 124
throw-expression, 124, 1267
throwing, see exception handling, throwing
<time.h>, 653
timed mutex types, 1224
token, 12
  alternative, 12
  preprocessing, 11
token, 12, 1259
traceable pointer object, 56, 433
trailing requires-clause, 179
trailing-return-type, 180, 1272
traits, 410
transfer ownership, 553
transform
  regular expression traits, 1159, 1196
transform_primary
  regular expression traits, 1160, 1196, 1197
TransformationTrait, 615
transition algorithm
  discard_block_engine, 964
  independent_bits_engine, 965
  linear_congruential_engine, 960
  mersenne_twister_engine, 961
  shuffle_order_engine, 965
  subtract_with_carry_engine, 962
translate
  regular expression traits, 1159, 1196
translate_nocase
  regular expression traits, 1160, 1196, 1197
TransformationTrait, 615
translation
  phases, 9–10
  separate, see compilation, separate
translation unit, 9, 46
  name and, 24
translation-unit, 46, 1263
trigraph sequence, 1296
trivial class, 211
trivial class type, 114
trivial type, 114
trivial types, 59
trivially copyable class, 211
trivially copyable types, 59
truncation, 78
try, 386
try block, see exception handling, try block
try-block, 386, 1277
<tuple>, 496
tuple
  and pair, 492
type, 24, 57–63
  allocated, 110
  arithmetic, 60
  promoted, 303
array, 61
bitmask, 413
Boolean, 59
char, 59
char16_t, 17, 20, 60, 63
char32_t, 17, 20, 60, 63
character, 59
character container, 408
class and, 211
compound, 61
const, 148
cv-combined, 82
cv-unqualified, 62
destination, 198
double, 60
dynamic, 3
enumerated, 61, 413
example of incomplete, 58
extended integer, 60
extended signed integer, 59
extended unsigned integer, 60
float, 60
floating-point, 59
function, 61, 187
fundamental, 59
implementation-defined sizeof, 59
incomplete, 26, 27, 30, 58, 76, 99–103, 109,
  110, 115, 125, 227
incompletely-defined object, 58
int, 59
integral, 59
  promoted, 303
long, 59
long double, 60
long long, 59
narrow character, 59
over-aligned, 57
pointer, 61
polymorphic, 232
referenceable, 409
short, 59
signed char, 59
signed integer, 59
similar, see similar types
standard integer, 60
standard signed integer, 59
standard unsigned integer, 60
static, 5
trivially copyable, 57
underlying
  char16_t, 60, 77
  char32_t, 60, 77
enumeration, 77, 157
fixed, 157
wchar_t, 60, 77
unsigned, 60
unsigned char, 59, 60
unsigned int, 60
unsigned integer, 60
unsigned long, 60
unsigned long long, 60
unsigned short, 60
void, 60
volatile, 148
wchar_t, 18, 20, 60, 63
type checking
argument, 101
type conversion, explicit, see casting
type generator, see template
type name, 180
nested, 223
scope of, 223
type pun, 107
type specifier
auto, 150, 153
bool, 150
char, 150
char16_t, 150
char32_t, 150
const, 148
decltype, 150
double, 150
elaborated, 44, 152
enum, 152
float, 150
int, 150
long, 150
short, 150
signed, 150
simple, 149
unsigned, 150
void, 150
volatile, 148, 149
wchar_t, 150
type-id, 180, 1273
type-name, 150, 1270
type-parameter, 308, 1276
type-parameter-key, 308, 1276
type-requirement, 97, 1265
type-specifier, 148, 1269
type-specifier-seq, 148, 1269
type_info, 104
<type_traits>, 615, 1319
typedef
function, 188
typedef
overloading and, 277
typedef-name, 143, 1269
typeid, 104
construction and, 267
destruction and, 267
<typeindex>, 653
<typeinfo>, 455
typename, 152
typename-specifier, 342, 1277
types
implementation-defined, 412

 BCH, 707
ud-suffix, 22, 1263
unary fold, 95
unary left fold, 95
unary operator
interpretation of, 300
overloaded, 300
unary right fold, 95
unary-expression, 108, 1266
unary-operator, 109, 1266
UnaryTypeTrait, 614
unblock, 5
undefined, 410, 427, 429, 430, 997, 1001, 1004, 1045
undefined behavior, see behavior, undefined
underlying type, see type, underlying
unevaluated operand, 83
Unicode required set, 405
uniform distributions, 970–972
uniform random bit generator
requirements, 952
uniform_int_distribution
– discrete probability function, 970
uniform_real_distribution
– probability density function, 971
union
– standard-layout, 212
union, 61, 224
– class versus, 211
– anonymous, 225
– global anonymous, 225
union-like class, 226
unique pointer, 552
unit
– translation, 417, 418, 428
universal character name, 9
universal-character-name, 10, 1259
unnamed bit-field, 222
unnamed-namespace-definition, 159, 1271
unordered associative containers, 772
begin, 781
bucket, 780
bucket_count, 780
bucket_size, 780
cbegin, 781
cend, 781
clear, 780
typeinfo, 781
complexity, 771
const_iterator, 773
find, 780
typeinfo, 771
complexity, 771
equal_range, 780
equality function, 771
equivalent keys, 772, 835, 843
erase, 780
exception safety, 782
find, 780
hash function, 771
hash_function, 775
hasier, 773
insert, 776, 777
iterator invalidation, 782
iterators, 782
key_eq, 775
key_equal, 773
key_type, 772
lack of comparison functions, 771
load_factor, 781
local_iterator, 773
max_bucket_count, 780
max_load_factor, 781
rehash, 781
requirements, 771, 772, 782
unique keys, 772, 828, 839
unordered_map, 827
unordered_map
element access, 833
unique keys, 828
unordered_multimap
equivalent keys, 835
unordered_multiset
equivalent keys, 843
unordered_set, 828
unordered_set
unique keys, 839
unqualified-id, 85, 1264
unsequenced, 65
unsigned
typedef and, 141
unsigned integer type, 60
unsigned-suffix, 16, 1261
unspecified, 450, 451, 456, 923, 1086, 1306, 1308
unspecified behavior, see behavior, unspecified
unwinding
stack, 388
uppercase, 14, 414
upstream, 587
upstream allocator, 584
user-defined literal, see literal, user-defined
overloaded, 302
user-defined-character-literal, 22, 1263
user-defined-floating-literal, 22, 1263
user-defined-integer-literal, 22, 1263
user-defined-literal, 22, 1263
user-defined-string-literal, 22, 1263
user-provided, 194
uses-allocator construction, 547
using-declaration, 163–168
using-declaration, 163, 1271
using-declarator, 163, 1271
using-declarator-list, 163, 1271
using-directive, 168–171
using-directive, 168, 1271
usual arithmetic conversions, see conversion, usual
arithmetic
usual deallocation function, 55
UTF-8 character literal, 17
UTF-8 string literal, 20
<utility>, 487, 1304
<valarray>, 987
valid, 29
valid but unspecified state, 410
value, 58
call by, 101
denormalized, see number, subnormal
indeterminate, 197
null member pointer, 79
null pointer, 78
undefined unrepresentable integral, 78
value category, 81
value computation, 64–65, 68, 102, 114, 123, 125,
126, 250
value representation, 58
value-initialization, 197
variable, 24
function-local predefined, 193
indeterminate uninitialized, 196
needed for constant evaluation, 129
program semantics affected by the existence
of a variable definition, 357
variable template
definition of, 306
variadic concept, see concept, variadic
<variant>, 518
variant member, 226
<vector>, 785
vectorization-unsafe, 904
virt-specifier, 215, 1274
virt-specifier-seq, 215, 1274
virtual base class, see base class, virtual
virtual function, 232–236
pure, 236, 237
virtual function call, 236
constructor and, 266
destructor and, 266
undefined pure, 237
visibility, 34
visible, 34
void*
type, 62
void&, 183
volatile, 62
constructor and, 220, 248
destructor and, 220, 255
implementation-defined, 149
overloading and, 277
volatile member function, 219
volatile-qualified, 62
waiting function, 1247
<wchar.h>, 706
wchar_t, see type, wchar_t
<wctype.h>, 704
weak result type, 1312
weakly parallel forward progress guarantees, 70
weibull_distribution
probability density function, 977
weights
  discrete_distribution, 983
  piecewise_constant_distribution, 984
weights at boundaries
  piecewise_linear_distribution, 985
well-formed program, see program, well-formed
white space, 12
wide string literal, 20
wide-character, 18
  null, 10
wide-character literal, 18
wide-character set
  basic execution, 10
  execution, 10
writable, 855

X(X&), see constructor, copy
xvalue, 80

Y_\ell^m (spherical associated Legendre functions), 1031

zero
  division by undefined, 80
  remainder undefined, 80
  undefined division by, 118
zero-initialization, 196
zeta functions \( \zeta \), 1030
## Index of grammar productions

The first page number for each entry is the page in the general text where the grammar production is defined. The second page number is the corresponding page in the Grammar summary (Annex A).

<table>
<thead>
<tr>
<th>Production</th>
<th>Page 1</th>
<th>Page 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract-declarator</td>
<td>180</td>
<td>1273</td>
</tr>
<tr>
<td>abstract-pack-declarator</td>
<td>180</td>
<td>1273</td>
</tr>
<tr>
<td>access-specifier</td>
<td>227</td>
<td>1275</td>
</tr>
<tr>
<td>additive-expression</td>
<td>119</td>
<td>1266</td>
</tr>
<tr>
<td>alias-declaration</td>
<td>139</td>
<td>1269</td>
</tr>
<tr>
<td>alignment-specifier</td>
<td>173</td>
<td>1271</td>
</tr>
<tr>
<td>and-expression</td>
<td>122</td>
<td>1267</td>
</tr>
<tr>
<td>asm-definition</td>
<td>171</td>
<td>1271</td>
</tr>
<tr>
<td>assignment-expression</td>
<td>125</td>
<td>1267</td>
</tr>
<tr>
<td>assignment-operator</td>
<td>125</td>
<td>1267</td>
</tr>
<tr>
<td>attribute</td>
<td>173</td>
<td>1271</td>
</tr>
<tr>
<td>attribute-argument-clause</td>
<td>174</td>
<td>1272</td>
</tr>
<tr>
<td>attribute-declaration</td>
<td>139</td>
<td>1269</td>
</tr>
<tr>
<td>attribute-list</td>
<td>173</td>
<td>1271</td>
</tr>
<tr>
<td>attribute-namespace</td>
<td>174</td>
<td>1272</td>
</tr>
<tr>
<td>attribute-scope-token</td>
<td>174</td>
<td>1272</td>
</tr>
<tr>
<td>attribute-specifier</td>
<td>173</td>
<td>1271</td>
</tr>
<tr>
<td>attribute-specifier-seq</td>
<td>173</td>
<td>1271</td>
</tr>
<tr>
<td>attribute-token</td>
<td>173</td>
<td>1272</td>
</tr>
<tr>
<td>attribute-using-prefix</td>
<td>173</td>
<td>1271</td>
</tr>
<tr>
<td>balanced-token</td>
<td>174</td>
<td>1272</td>
</tr>
<tr>
<td>balanced-token-seq</td>
<td>174</td>
<td>1272</td>
</tr>
<tr>
<td>base-clause</td>
<td>227</td>
<td>1275</td>
</tr>
<tr>
<td>base-specifier</td>
<td>227</td>
<td>1275</td>
</tr>
<tr>
<td>base-specifier-list</td>
<td>227</td>
<td>1275</td>
</tr>
<tr>
<td>binary-digit</td>
<td>15</td>
<td>1261</td>
</tr>
<tr>
<td>binary-exponent-part</td>
<td>19</td>
<td>1262</td>
</tr>
<tr>
<td>binary-literal</td>
<td>15</td>
<td>1261</td>
</tr>
<tr>
<td>block-declaration</td>
<td>139</td>
<td>1268</td>
</tr>
<tr>
<td>boolean-literal</td>
<td>21</td>
<td>1263</td>
</tr>
<tr>
<td>brace-or-equal-initializer</td>
<td>196</td>
<td>1273</td>
</tr>
<tr>
<td>braced-init-list</td>
<td>196</td>
<td>1273</td>
</tr>
<tr>
<td>c-char</td>
<td>17</td>
<td>1261</td>
</tr>
<tr>
<td>c-char-sequence</td>
<td>17</td>
<td>1261</td>
</tr>
<tr>
<td>capture</td>
<td>91</td>
<td>1264</td>
</tr>
<tr>
<td>capture-default</td>
<td>91</td>
<td>1264</td>
</tr>
<tr>
<td>capture-list</td>
<td>91</td>
<td>1264</td>
</tr>
<tr>
<td>cast-expression</td>
<td>117</td>
<td>1266</td>
</tr>
<tr>
<td>character-literal</td>
<td>17</td>
<td>1261</td>
</tr>
<tr>
<td>class-head</td>
<td>211</td>
<td>1274</td>
</tr>
<tr>
<td>class-head-name</td>
<td>211</td>
<td>1274</td>
</tr>
<tr>
<td>class-key</td>
<td>211</td>
<td>1274</td>
</tr>
<tr>
<td>class-name</td>
<td>211</td>
<td>1274</td>
</tr>
<tr>
<td>class-or-decltype</td>
<td>227</td>
<td>1275</td>
</tr>
<tr>
<td>class-specifier</td>
<td>211</td>
<td>1274</td>
</tr>
<tr>
<td>class-virt-specifier</td>
<td>211</td>
<td>1274</td>
</tr>
<tr>
<td>compare-expression</td>
<td>120</td>
<td>1266</td>
</tr>
<tr>
<td>compound-requirement</td>
<td>97</td>
<td>1265</td>
</tr>
<tr>
<td>compound-statement</td>
<td>131</td>
<td>1268</td>
</tr>
<tr>
<td>concept-definition</td>
<td>306</td>
<td>1276</td>
</tr>
<tr>
<td>concept-name</td>
<td>306</td>
<td>1276</td>
</tr>
<tr>
<td>condition</td>
<td>130</td>
<td>1267</td>
</tr>
<tr>
<td>conditional-expression</td>
<td>123</td>
<td>1267</td>
</tr>
<tr>
<td>conditionally-supported-directive</td>
<td>395</td>
<td>1278</td>
</tr>
<tr>
<td>constant-expression</td>
<td>126</td>
<td>1267</td>
</tr>
<tr>
<td>constrained-parameter</td>
<td>308</td>
<td>1276</td>
</tr>
<tr>
<td>constraint-expression</td>
<td>320</td>
<td>1276</td>
</tr>
<tr>
<td>constraint-logical-and-expression</td>
<td>306</td>
<td>1276</td>
</tr>
<tr>
<td>constraint-logical-or-expression</td>
<td>306</td>
<td>1276</td>
</tr>
<tr>
<td>control-line</td>
<td>395</td>
<td>1277</td>
</tr>
<tr>
<td>conversion-declarator</td>
<td>254</td>
<td>1275</td>
</tr>
<tr>
<td>conversion-function-id</td>
<td>254</td>
<td>1275</td>
</tr>
<tr>
<td>conversion-type-id</td>
<td>254</td>
<td>1275</td>
</tr>
<tr>
<td>ctor-initializer</td>
<td>260</td>
<td>1275</td>
</tr>
<tr>
<td>cv-qualifier</td>
<td>180</td>
<td>1272</td>
</tr>
<tr>
<td>cv-qualifier-seq</td>
<td>180</td>
<td>1272</td>
</tr>
<tr>
<td>d-char</td>
<td>20</td>
<td>1263</td>
</tr>
<tr>
<td>d-char-sequence</td>
<td>20</td>
<td>1263</td>
</tr>
<tr>
<td>decimal-floating-literal</td>
<td>18</td>
<td>1262</td>
</tr>
<tr>
<td>decimal-literal</td>
<td>15</td>
<td>1261</td>
</tr>
<tr>
<td>decl-specifier</td>
<td>141</td>
<td>1269</td>
</tr>
<tr>
<td>decl-specifier-seq</td>
<td>141</td>
<td>1269</td>
</tr>
<tr>
<td>declaration</td>
<td>139</td>
<td>1268</td>
</tr>
<tr>
<td>declaration-seq</td>
<td>139</td>
<td>1268</td>
</tr>
<tr>
<td>declaration-statement</td>
<td>137</td>
<td>1268</td>
</tr>
<tr>
<td>declarator</td>
<td>180</td>
<td>1272</td>
</tr>
<tr>
<td>declarator-id</td>
<td>180</td>
<td>1272</td>
</tr>
<tr>
<td>decltype-specifier</td>
<td>150</td>
<td>1270</td>
</tr>
<tr>
<td>deduction-guide</td>
<td>385</td>
<td>1277</td>
</tr>
<tr>
<td>default-template-argument</td>
<td>308</td>
<td>1276</td>
</tr>
<tr>
<td>defined-macro-expression</td>
<td>396</td>
<td>1278</td>
</tr>
<tr>
<td>defining-type-id</td>
<td>180</td>
<td>1273</td>
</tr>
<tr>
<td>defining-type-specifier</td>
<td>148</td>
<td>1269</td>
</tr>
<tr>
<td>defining-type-specifier-seq</td>
<td>148</td>
<td>1269</td>
</tr>
<tr>
<td>delete-expression</td>
<td>115</td>
<td>1266</td>
</tr>
<tr>
<td>designated-initializer-clause</td>
<td>196</td>
<td>1274</td>
</tr>
<tr>
<td>designated-initializer-list</td>
<td>196</td>
<td>1274</td>
</tr>
<tr>
<td>designator</td>
<td>196</td>
<td>1274</td>
</tr>
<tr>
<td>digit</td>
<td>13</td>
<td>1260</td>
</tr>
<tr>
<td>digit-sequence</td>
<td>19</td>
<td>1262</td>
</tr>
<tr>
<td>directory-separator</td>
<td>1117</td>
<td></td>
</tr>
<tr>
<td>elaborated-type-specifier</td>
<td>152</td>
<td>1270</td>
</tr>
<tr>
<td>else-group</td>
<td>395</td>
<td>1278</td>
</tr>
<tr>
<td>else-groups</td>
<td>395</td>
<td>1278</td>
</tr>
<tr>
<td>empty-declaration</td>
<td>139</td>
<td>1269</td>
</tr>
<tr>
<td>enclosing-namespace-specifier</td>
<td>159</td>
<td>1271</td>
</tr>
<tr>
<td>encoding-prefix</td>
<td>17</td>
<td>1261</td>
</tr>
</tbody>
</table>
Index of grammar productions
nonzero-digit, 15, 1261
noptr-abstract-declarator, 180, 1273
noptr-abstract-pack-declarator, 181, 1273
noptr-declarator, 180, 1272
noptr-new-declarator, 111, 1266
octal-digit, 15, 1261
octal-escape-sequence, 17, 1262
octal-literal, 15, 1261
opaque-enum-declaration, 156, 1270
operator, 299, 1275
operator-function-id, 299, 1275
parameter-declaration, 187, 1273
parameter-declaration-clause, 187, 1273
parameter-declaration-list, 187, 1273
parameters-and-qualifiers, 180, 1272
partial-concept-id, 308, 1276
pathname, 1117
pm-expression, 117, 1266
pointer-literal, 21, 1263
postfix-expression, 99, 1265
pp-number, 13, 1260
pp-tokens, 396, 1278
preferred-separator, 1117
preprocessing-file, 395, 1277
preprocessing-op-or-punc, 15, 1260
preprocessing-token, 11, 1259
primary-expression, 84, 1263
pseudo-destructor-name, 99, 1265
ptr-abstract-declarator, 180, 1273
ptr-declarator, 180, 1272
ptr-operator, 180, 1272
pure-specifier, 215, 1274
q-char, 12, 1260
q-char-sequence, 12, 1260
qualified-concept-name, 308, 1276
qualified-id, 86, 1264
qualified-name-space-specifier, 162, 1271
r-char, 20, 1263
r-char-sequence, 19, 1262
raw-string, 19, 1262
ref-qualifier, 180, 1272
relational-expression, 120, 1267
relative-path, 1117
replacement-list, 396, 1278
requirement, 96, 1265
requirement-body, 96, 1265
requirement-parameter-list, 96, 1265
requirement-seq, 96, 1265
requires-clause, 306, 1276
requires-expression, 96, 1265
return-type-requirement, 97, 1265
root-directory, 1117
root-name, 1117
s-char, 19, 1262
s-char-sequence, 19, 1262
selection-statement, 131, 1268
shift-expression, 119, 1266
sign, 19, 1262
simple-capture, 91, 1264
simple-declaration, 139, 1269
simple-escape-sequence, 17, 1262
simple-requirement, 97, 1265
simple-requirement, 97, 1265
statement, 130, 1267
statement-seq, 131, 1268
static_assert-declaration, 139, 1269
storage-class-specifier, 141, 1269
string-literal, 19, 1262
template-argument, 312, 1276
template-argument-list, 312, 1276
template-declaration, 306, 1275
template-head, 306, 1275
template-id, 312, 1276
template-name, 312, 1276
template-parameter, 307, 1276
template-parameter-list, 306, 1276
text-line, 395, 1278
throw-expression, 124, 1267
token, 12, 1259
trailing-return-type, 180, 1272
translation-unit, 46, 1263
try-block, 386, 1277
type-id, 180, 1273
type-name, 150, 1270
type-parameter, 308, 1276
type-parameter-key, 308, 1276
type-requirement, 97, 1265
type-specifier, 148, 1269
type-specifier-seq, 148, 1269
typedef-name, 143, 1269
typename-specifier, 342, 1277
ud-suffix, 22, 1263
unary-expression, 108, 1266
unary-operator, 109, 1266
universal-character-name, 10, 1259
unnamed-namespace-definition, 159, 1271
unqualified-id, 85, 1264
unsigned-suffix, 16, 1261
user-defined-character-literal, 22, 1263
user-defined-floating-literal, 22, 1263
user-defined-integer-literal, 22, 1263
user-defined-literal, 22, 1263
using-declaration, 163, 1271
using-declarator, 163, 1271
using-declarator-list, 163, 1271
using-directive, 168, 1271
virt-specifier, 215, 1274
virt-specifier-seq, 215, 1274

Index of grammar productions
Index of library headers

<algorithm>, 884
<any>, 529
<array>, 783
<assert.h>, 417, 475, 1301
<atomic>, 1198

<bitset>, 533
<cassert>, 417, 475, 748, 1301
<ccomplex>, 1301, 1303
<cctype>, 704
<cerrno>, 428, 476
<cfenv>, 941
<charconv>, 656
<chrono>, 110, 1301
<ccomplex>, 1301, 1303
<cassert>, 417, 475, 748, 1301
<cmath>, 1016, 1025
<codecvt>, 1322
<compare>, 120, 462
<complex>, 942
<complex.h>, 1301
<condition_variable>, 1239
<csetjmp>, 428, 469, 470, 1302
<csignal>, 469, 470
<cstdlib>, 1301, 1303
<ctime>, 187, 428, 469
<ctgmath>, 1301, 1303
<cwchar>, 428, 706, 1301
<cwctype>, 428, 704
<deque>, 783
<errno.h>, 476
<exception>, 457, 1312
<execversion>, 655
<filesystem>, 1110
<forward_list>, 784
<fstream>, 1091
<iostream>, 1301
<functional>, 594, 1316
<future>, 1245
<intializer_list>, 461
<inttypes.h>, 1157
<iomanip>, 1060
<iostream>, 1301
<locale>, 708, 709
<locale.h>, 749
<limits>, 438
<list>, 784
<map>, 1301
<map.h>, 1301
<numeric>, 1005
<optional>, 505
<ostream>, 1060
<queue>, 847
<random>, 957
<ratio>, 636
<regex>, 1160
<scoped_allocator>, 588
<set>, 810
<setjmp.h>, 470
<shared_mutex>, 1221
<string.h>, 470
<string_iterator>, 1082
<stack>, 848
<stdalign.h>, 1301, 1303
<stdarg.h>, 470
<stdatomic.h>, 416, 1301
<stdbool.h>, 1301, 1303
<stddef.h>, 18, 20
<stdexcept>, 472
<stdio.h>, 1156
<stdlib.h>, 1304
<stdnoreturn.h>, 416, 1301
<streambuf>, 1052
<string.h>, 665
<string.h>, 705
<string_view.h>, 695
<strstream>, 1305
<sstream>, 1104
<system_error>, 477
<tgmath.h>, 1301
<thread>, 1216
<threads.h>, 416, 1301
<time.h>, 653
<tuple>, 496
<type_traits>, 615, 1319
<typeindex>, 653
<typeinfo>, 455

<ustring.h>, 707
<unordered_map>, 827
<unordered_set>, 828
<utility>, 487, 1304

<valarray>, 987
<variant>, 518
<vector>, 785

<wchar.h>, 706
<wctype.h>, 704
# Index of library names

<table>
<thead>
<tr>
<th>Function/Type</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Exit</td>
<td>435, 447</td>
</tr>
<tr>
<td>_IOFBF</td>
<td>1154</td>
</tr>
<tr>
<td>_IOLBF</td>
<td>1154</td>
</tr>
<tr>
<td>_IONBF</td>
<td>1154</td>
</tr>
<tr>
<td>__alignas_is_defined</td>
<td>1303</td>
</tr>
<tr>
<td>__bool_true_false_are_defined</td>
<td>1303</td>
</tr>
<tr>
<td>_1</td>
<td>607</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>cauchy_distribution</td>
<td>981</td>
</tr>
<tr>
<td>extreme_value_distribution</td>
<td>978</td>
</tr>
<tr>
<td>uniform_int_distribution</td>
<td>971</td>
</tr>
<tr>
<td>uniform_real_distribution</td>
<td>972</td>
</tr>
<tr>
<td>weibull_distribution</td>
<td>977</td>
</tr>
<tr>
<td>abort</td>
<td>74, 136, 417, 435, 447, 454, 459</td>
</tr>
<tr>
<td>abs</td>
<td>435, 1016, 1025, 1156</td>
</tr>
<tr>
<td>complex</td>
<td>948</td>
</tr>
<tr>
<td>duration</td>
<td>649</td>
</tr>
<tr>
<td>valarray</td>
<td>997</td>
</tr>
<tr>
<td>absolute</td>
<td>1142</td>
</tr>
<tr>
<td>accumulate</td>
<td>1008</td>
</tr>
<tr>
<td>acos</td>
<td>1016</td>
</tr>
<tr>
<td>complex</td>
<td>948</td>
</tr>
<tr>
<td>valarray</td>
<td>997</td>
</tr>
<tr>
<td>acosf</td>
<td>1016</td>
</tr>
<tr>
<td>acosh</td>
<td>1016</td>
</tr>
<tr>
<td>complex</td>
<td>948</td>
</tr>
<tr>
<td>acoshf</td>
<td>1016</td>
</tr>
<tr>
<td>acoshl</td>
<td>1016</td>
</tr>
<tr>
<td>acosl</td>
<td>1016</td>
</tr>
<tr>
<td>acq_rel</td>
<td></td>
</tr>
<tr>
<td>memory_order</td>
<td>1201</td>
</tr>
<tr>
<td>acquire</td>
<td></td>
</tr>
<tr>
<td>memory_order</td>
<td>1201</td>
</tr>
<tr>
<td>add_const</td>
<td>631</td>
</tr>
<tr>
<td>add_cv</td>
<td>631</td>
</tr>
<tr>
<td>add_lvalue_reference</td>
<td>631</td>
</tr>
<tr>
<td>add_pointer</td>
<td>633</td>
</tr>
<tr>
<td>add_rvalue_reference</td>
<td>631</td>
</tr>
<tr>
<td>add_volatile</td>
<td>631</td>
</tr>
<tr>
<td>address</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>1317</td>
</tr>
<tr>
<td>addressof</td>
<td>550</td>
</tr>
<tr>
<td>adjacent_difference</td>
<td>1015</td>
</tr>
<tr>
<td>adjacent_find</td>
<td>908</td>
</tr>
<tr>
<td>adopt_lock</td>
<td>1229</td>
</tr>
<tr>
<td>adopt_lock_t</td>
<td>1229</td>
</tr>
<tr>
<td>advance</td>
<td>865</td>
</tr>
<tr>
<td>align</td>
<td>546</td>
</tr>
<tr>
<td>align_val_t</td>
<td>448</td>
</tr>
<tr>
<td>aligned_alloc</td>
<td>435, 552, 1302</td>
</tr>
<tr>
<td>aligned_storage</td>
<td>633, 634</td>
</tr>
<tr>
<td>aligned_union</td>
<td>633</td>
</tr>
<tr>
<td>alignment_of</td>
<td>628</td>
</tr>
<tr>
<td>all</td>
<td></td>
</tr>
<tr>
<td>bitset</td>
<td>538</td>
</tr>
<tr>
<td>all_of</td>
<td>905</td>
</tr>
<tr>
<td>allocate</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>549, 1317</td>
</tr>
<tr>
<td>allocator_traits</td>
<td>548</td>
</tr>
<tr>
<td>memory_resource</td>
<td>580</td>
</tr>
<tr>
<td>polymorphic_allocator</td>
<td>581</td>
</tr>
<tr>
<td>scoped_allocator_adaptor</td>
<td>591</td>
</tr>
<tr>
<td>allocate_shared</td>
<td>567–569</td>
</tr>
<tr>
<td>allocator</td>
<td>549, 1317</td>
</tr>
<tr>
<td>address</td>
<td>1317</td>
</tr>
<tr>
<td>allocate</td>
<td>549, 1317</td>
</tr>
<tr>
<td>construct</td>
<td>1318</td>
</tr>
<tr>
<td>deallocate</td>
<td>549</td>
</tr>
<tr>
<td>destroy</td>
<td>1318</td>
</tr>
<tr>
<td>is_always_equal</td>
<td>549</td>
</tr>
<tr>
<td>max_size</td>
<td>1318</td>
</tr>
<tr>
<td>operator!</td>
<td>550</td>
</tr>
<tr>
<td>operator==</td>
<td>550</td>
</tr>
<tr>
<td>propagate_on_container_move_assignment</td>
<td></td>
</tr>
<tr>
<td>value_type</td>
<td>549</td>
</tr>
<tr>
<td>allocator_arg</td>
<td>546</td>
</tr>
<tr>
<td>allocator_arg_t</td>
<td>546</td>
</tr>
<tr>
<td>allocator_traits</td>
<td>547</td>
</tr>
<tr>
<td>allocate</td>
<td>548</td>
</tr>
<tr>
<td>const_pointer</td>
<td>548</td>
</tr>
<tr>
<td>const_void_pointer</td>
<td>548</td>
</tr>
<tr>
<td>construct</td>
<td>549</td>
</tr>
<tr>
<td>deallocate</td>
<td>548</td>
</tr>
<tr>
<td>destroy</td>
<td>549</td>
</tr>
<tr>
<td>difference_type</td>
<td>548</td>
</tr>
<tr>
<td>is_always_equal</td>
<td>548</td>
</tr>
<tr>
<td>max_size</td>
<td>549</td>
</tr>
<tr>
<td>pointer</td>
<td>548</td>
</tr>
<tr>
<td>propagate_on_container_copy_assignment</td>
<td></td>
</tr>
<tr>
<td>propagate_on_container_move_assignment</td>
<td></td>
</tr>
<tr>
<td>size_type</td>
<td>548</td>
</tr>
<tr>
<td>void_pointer</td>
<td>548</td>
</tr>
<tr>
<td>allocator_type</td>
<td></td>
</tr>
<tr>
<td>basic_string</td>
<td>669</td>
</tr>
<tr>
<td>alpha</td>
<td></td>
</tr>
</tbody>
</table>
gamma_distribution, 977
always_noconv
codecvt, 722
any
constructor, 530, 531
destructor, 531
emplace, 531, 532
has_value, 532
operator=, 531
reset, 532
swap, 532
type, 532
any (member)
bitset, 538
any_cast, 532, 533
any_of, 905
append
basic_string, 679, 680
path, 1122
apply, 502
valarray, 996
arg, 949
complex, 948
argument_type
bit_not, 1313
function, 1313
hash, 1315
logical_not, 1313
mem_fn, 1315
negate, 1313
reference_wrapper, 1315
unary_negate, 1316
array, 785, 787
begin, 785
data, 787
end, 785
fill, 787
get, 787
max_size, 785
size, 785, 787
swap, 787
as_const, 491
asctime, 653
asin, 1016
complex, 948
valarray, 997
asinf, 1016
asinh, 1016
complex, 948
asinhf, 1016
asinhl, 1016
asinl, 1016
assert, 475
assign
basic_regex, 1175
basic_string, 680, 681
directory_entry, 1135
error_code, 482
error_condition, 484
path, 1121
assoc_laguerre, 1026
assoc_laguerref, 1026
assoc_laguerrel, 1026
assoc_legendre, 1026
assoc_legendref, 1026
assoc_legendrel, 1026
async, 1255
at
basic_string, 678
basic_string_view, 699
map, 815
unordered_map, 833
at_quick_exit, 417, 435, 448
atan, 1016
complex, 948
valarray, 997
atan2, 1016
valarray, 997
atan2f, 1016
atan2l, 1016
atanf, 1016
atanhl, 1016
atanlh, 1016
complex, 948
atanhf, 1016
atanhl, 1016
atanh, 1016
atexit, 73, 417, 435, 447
atof, 435
atoi, 435
atoll, 435
atomic, 1204
compare_exchange_strong, 1205
compare_exchange_weak, 1205
constructor, 1204
exchange, 1205
is_always_lock_free, 1205
is_lock_free, 1205
load, 1205
operator type, 1205
operator=, 1205
store, 1205
value_type, 1204
atomic<floating-point>, 1208
compare_exchange_strong, 1205
compare_exchange_weak, 1205
constructor, 1204
exchange, 1205
fetch_add, 1209
fetch_sub, 1209
is_always_lock_free, 1205
is_lock_free, 1205
load, 1205
operator floating-point, 1205
operator+=, 1209
operator-=, 1209
operator/=, 1205
store, 1205
atomic<integral>, 1207
  compare_exchange_strong, 1205
  compare_exchange_weak, 1205
  constructor, 1204
  exchange, 1205
  fetch_add, 1208
  fetch_and, 1208
  fetch_or, 1208
  fetch_sub, 1208
  fetch_xor, 1208
  is_always_lock_free, 1205
  is_lock_free, 1205
  load, 1205
  operator integral, 1205
  operator++, 1211
  operator+, 1208
  operator-, 1208
  operator--, 1211
  operator=, 1205
  operator&=, 1208
  operator^=, 1208
  operator|, 1208
  store, 1205

atomic<shared_ptr<T>>, 575
  compare_exchange_strong, 576, 577
  compare_exchange_weak, 576, 577
  constructor, 576
  exchange, 576
  load, 576
  operator shared_ptr<T>, 576
  operator=, 576
  store, 576

atomic<T*>, 1210, 1211
  compare_exchange_strong, 1205
  compare_exchange_weak, 1205
  constructor, 1204
  exchange, 1205
  fetch_add, 1211
  fetch_sub, 1211
  is_always_lock_free, 1205
  is_lock_free, 1205
  load, 1205
  operator T*, 1205
  operator++, 1211
  operator+, 1208, 1209, 1211
  operator-, 1208, 1209, 1211
  operator--, 1211
  operator=, 1205
  store, 1205

atomic<weak_ptr<T>>, 577
  compare_exchange_strong, 578
  compare_exchange_weak, 578
  constructor, 577
  exchange, 578
  load, 578
  operator weak_ptr<T>, 578
  operator=, 578
  store, 578

atomic_bool, 1201

ATOMIC_BOOL_LOCK_FREE, 1203
atomic_char, 1201
atomic_char16_t, 1201
ATOMIC_CHAR16_T_LOCK_FREE, 1203
atomic_char32_t, 1201
ATOMIC_CHAR32_T_LOCK_FREE, 1203
ATOMIC_CHAR_LOCK_FREE, 1203
atomic_compare_exchange_strong, 1205
  shared_ptr, 1322
atomic_compare_exchange_strong_explicit, 1205
  shared_ptr, 1322
atomic_compare_exchange_weak, 1205
  shared_ptr, 1322
atomic_compare_exchange_weak_explicit, 1205
  shared_ptr, 1322
atomic_exchange, 1205
  shared_ptr, 1322
atomic_exchange_explicit, 1205
  shared_ptr, 1322
atomic_fetch_add, 1208, 1209, 1211
atomic_fetch_add_explicit, 1208, 1209, 1211
atomic_fetch_or, 1208
atomic_fetch_or_explicit, 1208
atomic_fetch_xor, 1208
atomic_fetch_xor_explicit, 1208
atomic_flag
  clear, 1212
  test_and_set, 1212
  atomic_flag_clear, 1212
atomic_flag_clear_explicit, 1212
atomic_flag_test_and_set, 1212
atomic_flag_test_and_set_explicit, 1212
atomic_init, 1211
atomic_int, 1201
atomic_int16_t, 1201
atomic_int32_t, 1201
atomic_int64_t, 1201
atomic_int8_t, 1201
atomic_int_fast16_t, 1201
atomic_int_fast32_t, 1201
atomic_int_fast64_t, 1201
atomic_int_fast8_t, 1201
atomic_int_least16_t, 1201
atomic_int_least32_t, 1201
atomic_int_least64_t, 1201
atomic_int_least8_t, 1201
ATOMIC_INT_LOCK_FREE, 1203
atomic_intmax_t, 1201
atomic_intptr_t, 1201
atomic_is_lock_free, 1205
  shared_ptr, 1321
atomic_llvm, 1205
ATOMIC_LLONG_LOCK_FREE, 1203
Index of library names

atomic_load, 1205
    shared_ptr, 1321
atomic_load_explicit, 1205
    shared_ptr, 1321
atomic_long, 1201
ATOMIC_LONG_LOCK_FREE, 1203
ATOMIC_POINTER_LOCK_FREE, 1203
atomic_ptrdiff_t, 1201
atomic_schar, 1201
atomic_short, 1201
ATOMIC_SHORT_LOCK_FREE, 1203
atomic_signal_fence, 1213
atomic_size_t, 1201
atomic_store, 1205
    shared_ptr, 1321
atomic_store_explicit, 1205
    shared_ptr, 1322
atomic_thread_fence, 1213
atomic_uchar, 1201
atomic_uint, 1201
atomic_uint16_t, 1201
atomic_uint32_t, 1201
atomic_uint8_t, 1201
    atomic_uint_fast16_t, 1201
    atomic_uint_fast32_t, 1201
    atomic_uint_fast64_t, 1201
    atomic_uint_fast8_t, 1201
    atomic_uint_least16_t, 1201
    atomic_uint_least32_t, 1201
    atomic_uint_least64_t, 1201
    atomic_uint_least8_t, 1201
atomic_uintmax_t, 1201
atomic_ullong, 1201
atomic_ushort, 1201
ATOMIC_VAR_INIT, 1205
atomic_wchar_t, 1201
    ATOMIC_WCHAR_T_LOCK_FREE, 1203
auto_ptr
    zombie, 427

b
    cauchy_distribution, 981
    extreme_value_distribution, 978
    uniform_int_distribution, 971
    uniform_real_distribution, 972
    weibull_distribution, 977
back
    basic_string, 678
    basic_string_view, 699
back_insert_iterator, 869
    constructor, 870
    operator*, 870
    operator++, 870
    operator=, 870
back_inserter, 870
bad
    basic_ios, 1049
    bad_alloc, 114, 449, 453, 454
        constructor, 453
        operator=, 453
        what, 454
    bad_any_cast, 529
        what, 529
    bad_array_new_length, 454
        constructor, 454
        what, 454
    bad_cast, 103, 455, 456
        constructor, 456, 457
        operator=, 457
        what, 457
    bad_exception, 458
        constructor, 458
        operator=, 458
        what, 459
    bad_function_call, 608
        constructor, 608
        what, 608
    bad_optional_access
        constructor, 515
        what, 515
    bad_typeid, 104, 455, 457
        constructor, 457
        operator=, 457
        what, 457
    bad_variant_access, 528
        constructor, 528
        what, 528
    bad_weak_ptr, 561
        constructor, 562
        what, 562
base
    move_iterator, 874
    raw_storage_iterator, 1319
    reverse_iterator, 867
basic_filebuf, 1033, 1092
    close, 1095
    constructor, 1093
    destructor, 1094
    imbue, 1097
    is_open, 1094
    open, 1094
    operator=, 1094
    overflow, 1096
    pbackfail, 1096
    seekoff, 1097
    seekpos, 1097
    setbuf, 1096
    showmanyc, 1095
    swap, 1094
    sync, 1097
    uflow, 1096
    underflow, 1095
basic_filebuf<char>, 1091
basic_filebuf<wchar_t>, 1091
basic_fstream, 1033, 1102
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>1104</td>
</tr>
<tr>
<td>constructor</td>
<td>1103</td>
</tr>
<tr>
<td>is_open</td>
<td>1103</td>
</tr>
<tr>
<td>open</td>
<td>1103, 1104</td>
</tr>
<tr>
<td>operator=</td>
<td>1103</td>
</tr>
<tr>
<td>rdbuf</td>
<td>1103</td>
</tr>
<tr>
<td>swap</td>
<td>1103</td>
</tr>
<tr>
<td>basic_fstream&lt;char&gt;</td>
<td>1091</td>
</tr>
<tr>
<td>basic_fstream&lt;wchar_t&gt;</td>
<td>1091</td>
</tr>
<tr>
<td>basic_ifstream</td>
<td>1033, 1098</td>
</tr>
<tr>
<td>close</td>
<td>1099</td>
</tr>
<tr>
<td>constructor</td>
<td>1098, 1099</td>
</tr>
<tr>
<td>is_open</td>
<td>1099</td>
</tr>
<tr>
<td>open</td>
<td>1099</td>
</tr>
<tr>
<td>operator=</td>
<td>1099</td>
</tr>
<tr>
<td>rdbuf</td>
<td>1099</td>
</tr>
<tr>
<td>swap</td>
<td>1099</td>
</tr>
<tr>
<td>basic_ifstream&lt;char&gt;</td>
<td>1091</td>
</tr>
<tr>
<td>basic_ifstream&lt;wchar_t&gt;</td>
<td>1091</td>
</tr>
<tr>
<td>basic_ios</td>
<td>1033, 1045</td>
</tr>
<tr>
<td>bad</td>
<td>1049</td>
</tr>
<tr>
<td>clear</td>
<td>1049</td>
</tr>
<tr>
<td>constructor</td>
<td>1046</td>
</tr>
<tr>
<td>copyfmt</td>
<td>1048</td>
</tr>
<tr>
<td>destructor</td>
<td>1046</td>
</tr>
<tr>
<td>eof</td>
<td>1049</td>
</tr>
<tr>
<td>exceptions</td>
<td>1049</td>
</tr>
<tr>
<td>fail</td>
<td>1049</td>
</tr>
<tr>
<td>fill</td>
<td>1047</td>
</tr>
<tr>
<td>good</td>
<td>1049</td>
</tr>
<tr>
<td>imbue</td>
<td>1047</td>
</tr>
<tr>
<td>init</td>
<td>1047, 1062</td>
</tr>
<tr>
<td>move</td>
<td>1048</td>
</tr>
<tr>
<td>narrow</td>
<td>1047</td>
</tr>
<tr>
<td>operator bool</td>
<td>1049</td>
</tr>
<tr>
<td>operator!</td>
<td>1049</td>
</tr>
<tr>
<td>rdbuf</td>
<td>1047</td>
</tr>
<tr>
<td>rdstate</td>
<td>1049</td>
</tr>
<tr>
<td>set_rdstate</td>
<td>1048</td>
</tr>
<tr>
<td>setstate</td>
<td>1049</td>
</tr>
<tr>
<td>swap</td>
<td>1048</td>
</tr>
<tr>
<td>tie</td>
<td>1047</td>
</tr>
<tr>
<td>widen</td>
<td>1047</td>
</tr>
<tr>
<td>basic_ios&lt;char&gt;</td>
<td>1037</td>
</tr>
<tr>
<td>basic_ios&lt;wchar_t&gt;</td>
<td>1037</td>
</tr>
<tr>
<td>basic_iostream</td>
<td>1070</td>
</tr>
<tr>
<td>constructor</td>
<td>1071</td>
</tr>
<tr>
<td>destructor</td>
<td>1071</td>
</tr>
<tr>
<td>operator=</td>
<td>1071</td>
</tr>
<tr>
<td>swap</td>
<td>1071</td>
</tr>
<tr>
<td>basic_istream</td>
<td>1033, 1061</td>
</tr>
<tr>
<td>constructor</td>
<td>1062, 1063</td>
</tr>
<tr>
<td>destructor</td>
<td>1063</td>
</tr>
<tr>
<td>gcount</td>
<td>1066</td>
</tr>
<tr>
<td>get</td>
<td>1067</td>
</tr>
<tr>
<td>getline</td>
<td>1067, 1068</td>
</tr>
<tr>
<td>ignore</td>
<td>1068</td>
</tr>
<tr>
<td>operator=</td>
<td>1063</td>
</tr>
<tr>
<td>operator&gt;&gt;</td>
<td>1064–1066, 1070</td>
</tr>
<tr>
<td>peek</td>
<td>1069</td>
</tr>
<tr>
<td>putback</td>
<td>1069</td>
</tr>
<tr>
<td>read</td>
<td>1069</td>
</tr>
<tr>
<td>readsome</td>
<td>1069</td>
</tr>
<tr>
<td>seekg</td>
<td>1070</td>
</tr>
<tr>
<td>swap</td>
<td>1063</td>
</tr>
<tr>
<td>sync</td>
<td>1069</td>
</tr>
<tr>
<td>tellg</td>
<td>1070</td>
</tr>
<tr>
<td>unget</td>
<td>1069</td>
</tr>
<tr>
<td>basic_istream::sentry</td>
<td>1063</td>
</tr>
<tr>
<td>constructor</td>
<td>1063</td>
</tr>
<tr>
<td>destructor</td>
<td>1064</td>
</tr>
<tr>
<td>operator bool</td>
<td>1064</td>
</tr>
<tr>
<td>basic_istream&lt;char&gt;</td>
<td>1060</td>
</tr>
<tr>
<td>basic_istream&lt;wchar_t&gt;</td>
<td>1060</td>
</tr>
<tr>
<td>basic_istreambuf_iterator</td>
<td>1033</td>
</tr>
<tr>
<td>basic_istringstream</td>
<td>1033, 1087</td>
</tr>
<tr>
<td>constructor</td>
<td>1088</td>
</tr>
<tr>
<td>operator=</td>
<td>1088</td>
</tr>
<tr>
<td>rdbuf</td>
<td>1088</td>
</tr>
<tr>
<td>str</td>
<td>1088</td>
</tr>
<tr>
<td>swap</td>
<td>1088</td>
</tr>
<tr>
<td>basic_istringstream&lt;char&gt;</td>
<td>1082</td>
</tr>
<tr>
<td>basic_istringstream&lt;wchar_t&gt;</td>
<td>1082</td>
</tr>
<tr>
<td>basic_ofstream</td>
<td>1033, 1100</td>
</tr>
<tr>
<td>close</td>
<td>1101</td>
</tr>
<tr>
<td>constructor</td>
<td>1100, 1101</td>
</tr>
<tr>
<td>is_open</td>
<td>1101</td>
</tr>
<tr>
<td>open</td>
<td>1101</td>
</tr>
<tr>
<td>operator=</td>
<td>1101</td>
</tr>
<tr>
<td>rdbuf</td>
<td>1101</td>
</tr>
<tr>
<td>swap</td>
<td>1101</td>
</tr>
<tr>
<td>basic_ofstream&lt;char&gt;</td>
<td>1091</td>
</tr>
<tr>
<td>basic_ofstream&lt;wchar_t&gt;</td>
<td>1091</td>
</tr>
<tr>
<td>basic_ostream</td>
<td>1033, 1071, 1181</td>
</tr>
<tr>
<td>constructor</td>
<td>1073</td>
</tr>
<tr>
<td>destructor</td>
<td>1073</td>
</tr>
<tr>
<td>flush</td>
<td>1078</td>
</tr>
<tr>
<td>init</td>
<td>1073</td>
</tr>
<tr>
<td>operator&lt;&lt;</td>
<td>1075–1078</td>
</tr>
<tr>
<td>operator=</td>
<td>1073</td>
</tr>
<tr>
<td>put</td>
<td>1077</td>
</tr>
<tr>
<td>seekp</td>
<td>1074</td>
</tr>
<tr>
<td>swap</td>
<td>1073</td>
</tr>
<tr>
<td>tellp</td>
<td>1074</td>
</tr>
<tr>
<td>write</td>
<td>1077</td>
</tr>
<tr>
<td>basic_ostream::sentry</td>
<td>1073</td>
</tr>
<tr>
<td>constructor</td>
<td>1074</td>
</tr>
<tr>
<td>destructor</td>
<td>1074</td>
</tr>
<tr>
<td>basic_ostream&lt;char&gt;</td>
<td>1060</td>
</tr>
<tr>
<td>basic_ostream&lt;wchar_t&gt;</td>
<td>1060</td>
</tr>
<tr>
<td>basic_ostreambuf_iterator</td>
<td>1033</td>
</tr>
<tr>
<td>basic_ostreamstream</td>
<td>1033, 1088</td>
</tr>
<tr>
<td>constructor</td>
<td>1089</td>
</tr>
<tr>
<td>operator=</td>
<td>1089</td>
</tr>
<tr>
<td>rdbuf</td>
<td>1090</td>
</tr>
<tr>
<td>str</td>
<td>1090</td>
</tr>
<tr>
<td>swap</td>
<td>1090</td>
</tr>
<tr>
<td>Library Name</td>
<td>Section Numbers</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>basic_string</td>
<td>669, 688, 1083</td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1052</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>685</td>
</tr>
<tr>
<td>allocator_type</td>
<td>669</td>
</tr>
<tr>
<td>append</td>
<td>679, 680</td>
</tr>
<tr>
<td>assign</td>
<td>680, 681</td>
</tr>
<tr>
<td>at</td>
<td>678</td>
</tr>
<tr>
<td>back</td>
<td>678</td>
</tr>
<tr>
<td>begin</td>
<td>676</td>
</tr>
<tr>
<td>c_str</td>
<td>684</td>
</tr>
<tr>
<td>capacity</td>
<td>677</td>
</tr>
<tr>
<td>cbegin</td>
<td>676</td>
</tr>
<tr>
<td>cend</td>
<td>677</td>
</tr>
<tr>
<td>clear</td>
<td>678</td>
</tr>
<tr>
<td>compare</td>
<td>687, 688</td>
</tr>
<tr>
<td>const_iterator</td>
<td>669</td>
</tr>
<tr>
<td>const_pointer</td>
<td>669</td>
</tr>
<tr>
<td>const_reference</td>
<td>669</td>
</tr>
<tr>
<td>const_reverse_iterator</td>
<td>669</td>
</tr>
<tr>
<td>constructor</td>
<td>673–675</td>
</tr>
<tr>
<td>copy</td>
<td>684</td>
</tr>
<tr>
<td>crbegin</td>
<td>677</td>
</tr>
<tr>
<td>crend</td>
<td>677</td>
</tr>
<tr>
<td>data</td>
<td>684</td>
</tr>
<tr>
<td>difference_type</td>
<td>669</td>
</tr>
<tr>
<td>empty</td>
<td>678</td>
</tr>
<tr>
<td>end</td>
<td>677</td>
</tr>
<tr>
<td>ends_with</td>
<td>688</td>
</tr>
<tr>
<td>erase</td>
<td>682</td>
</tr>
<tr>
<td>find</td>
<td>685</td>
</tr>
<tr>
<td>find_first_not_of</td>
<td>686, 687</td>
</tr>
<tr>
<td>find_first_of</td>
<td>685, 686</td>
</tr>
<tr>
<td>find_last_not_of</td>
<td>687</td>
</tr>
<tr>
<td>find_last_of</td>
<td>686</td>
</tr>
<tr>
<td>front</td>
<td>678</td>
</tr>
<tr>
<td>get_allocator</td>
<td>685</td>
</tr>
<tr>
<td>getline</td>
<td>692</td>
</tr>
<tr>
<td>insert</td>
<td>681, 682</td>
</tr>
<tr>
<td>iterator</td>
<td>669</td>
</tr>
<tr>
<td>length</td>
<td>677</td>
</tr>
<tr>
<td>max_size</td>
<td>677</td>
</tr>
<tr>
<td>operator basic_string_view</td>
<td>685</td>
</tr>
<tr>
<td>operator!</td>
<td>690</td>
</tr>
<tr>
<td>operator*</td>
<td>688–690</td>
</tr>
<tr>
<td>operator*=</td>
<td>678, 679</td>
</tr>
<tr>
<td>operator&lt;</td>
<td>690</td>
</tr>
<tr>
<td>operator&lt;=</td>
<td>692</td>
</tr>
<tr>
<td>operator&lt;=&gt;</td>
<td>691</td>
</tr>
<tr>
<td>operator==</td>
<td>676</td>
</tr>
<tr>
<td>operator==</td>
<td>690</td>
</tr>
<tr>
<td>operator&gt;=</td>
<td>690, 691</td>
</tr>
<tr>
<td>operator&gt;&gt;</td>
<td>691</td>
</tr>
<tr>
<td>operator[]</td>
<td>678</td>
</tr>
<tr>
<td>pointer</td>
<td>669</td>
</tr>
<tr>
<td>pop_back</td>
<td>682</td>
</tr>
<tr>
<td>push_back</td>
<td>680</td>
</tr>
<tr>
<td>rbegin</td>
<td>682</td>
</tr>
<tr>
<td>reference</td>
<td>669</td>
</tr>
</tbody>
</table>
rend, 677
replace, 682–684
reserve, 677
resize, 677
reverse_iterator, 669
rfind, 685
shrink_to_fit, 678
size, 677
size_type, 669
starts_with, 688
substr, 687
swap, 684, 691
traits_type, 669
value_type, 669

basic_string_view, 695
at, 699
back, 699
begin, 698
cbegin, 698
cend, 698
compare, 700
const_iterator, 695, 698
const_pointer, 695
const_reference, 695
const_reverse_iterator, 695
constructor, 697, 698
copy, 699
crbegin, 698
crend, 698
data, 699
difference_type, 695
empty, 698
end, 698
ends_with, 700
find, 701
find_first_not_of, 701
find_first_of, 701
find_last_not_of, 702
find_last_of, 701
front, 699
iterator, 695
length, 698
max_size, 698
operator!=, 703
operator<, 703
operator<<, 703
operator<=, 703
operator==, 702
operator>, 703
operator>=, 703
operator[], 698
pointer, 695
rbegin, 698
reference, 695
remove_prefix, 699
remove_suffix, 699
rend, 698
reverse_iterator, 695
rfind, 701
size, 698
size_type, 695
starts_with, 700
substr, 699
swap, 699
traits_type, 695
value_type, 695

basic_stringbuf, 1033, 1083
constructor, 1084
operator=, 1085
overflow, 1086
pbackfail, 1085
seekoff, 1086
seekpos, 1087
str, 1085
swap, 1085
underflow, 1085
basic_stringbuf<char>, 1082
basic_stringbuf<wchar_t>, 1082
basic_stringstream, 1033, 1090
constructor, 1091
operator=, 1091
rdbuf, 1091
str, 1091
swap, 1091
basic_stringstream<char>, 1082
basic_stringstream<wchar_t>, 1082
basic_syncbuf, 1033, 1104
constructor, 1105
destructor, 1105
emit, 1106
get_allocator, 1106
get_wrapped, 1106
operator=, 1105
set_emit_on_sync, 1106
swap, 1106
csync, 1106

before
type_info, 456
before_begin
forward_list, 794
begin, 461
array, 785
basic_string, 676
basic_string_view, 698
directory_iterator, 1139
initializer_list, 462
match_results, 1184
path, 1127
recursive_directory_iterator, 1142
valarray, 1004
begin(C&), 882
begin(initializer_list<E>), 462
begin(T (&)[N]), 882
bernoulli_distribution, 972
constructor, 972
p, 972
beta, 1026
gamma_distribution, 977
Index of library names
call_once, 1238
calloc, 435, 552, 1302
canonical, 1143
capacity
  - basic_string, 677
  - vector, 805
casin
  - complex, 948
casinh
  - complex, 948
catan
  - complex, 948
catanh
  - complex, 948
category
  - error_code, 482
  - error_condition, 484
  - locale, 711
ciauchy_distribution, 980, 981
  - a, 981
  - b, 981
  - constructor, 981
cbegin
  - forward_list, 794
cbegin
  - basic_string, 676
  - basic_string_view, 698
cbegin(const C&), 882
cbrtf, 1016
cbrtl, 1016
cceil, 1016
  - duration, 648
  - time_point, 651
cceilf, 1016
cceill, 1016
cend
  - basic_string, 677
  - basic_string_view, 698
cend(const C&), 882
cerr, 1036
CHAR_BIT, 445
char_class_type
  - regex_traits, 1169
CHAR_MAX, 445
CHAR_MIN, 445
char_traits, 662–664
  - char_type, 662
  - int_type, 662
  - off_type, 662
  - pos_type, 662
  - state_type, 662
char_type
  - char_traits, 662
chars_format, 656
  - fixed, 656
  - general, 656
  - hex, 656
  - scientific, 656
chi_squared_distribution, 980
  - constructor, 980
  - n, 980
chrono, 639
cin, 1036
clamp, 936
classic
  - locale, 715
classic_table
  - ctype<char>, 721
clear
  - atomic_flag, 1212
  - basic_ios, 1049
  - basic_string, 678
  - error_code, 482
  - error_condition, 484
  - forward_list, 795
  - path, 1122
clearerr, 1154
clock, 653
clock_t, 653
CLOCKS_PER_SEC, 653
clog, 1036
close
  - basic_filebuf, 1095
  - basic_fstream, 1104
  - basic_ifstream, 1099
  - basic_ofstream, 1101
  - messages, 745
code
  - future_error, 1247
  - system_error, 486
codecvt, 721
  - always_noconv, 722
  - do_always_noconv, 724
  - do_encoding, 724
  - do_in, 723
  - do_length, 724
  - do_max_length, 724
  - do_out, 723
  - do_unshift, 724
  - encoding, 722
  - in, 722
  - length, 722
  - max_length, 722
  - out, 722
  - unshift, 722
codecvt_byname, 724
codecvt_mode, 1323
codecvt_utf16, 1323
codecvt_utf8, 1323
codecvt_utf8_utf16, 1323
collate, 733
  - compare, 734
  - do_compare, 734
  - do_hash, 734
  - do_transform, 734
  - hash, 734
  - transform, 734
collate_byname, 734
combine
locale, 714
common_comparison_category, 468
common_comparison_category_t, 462
common_type, 634, 643, 646
comp_ellint_1, 1027
comp_ellint_1f, 1027
comp_ellint_1l, 1027
comp_ellint_2, 1027
comp_ellint_2f, 1027
comp_ellint_2l, 1027
comp_ellint_3, 1027
comp_ellint_3f, 1027
comp_ellint_3l, 1027
compare
basic_string, 687, 688
basic_string_view, 700
collate, 734
path, 1124, 1125
sub_match, 1177
compare_3way, 937
compare_exchange_strong
atomic, 1205
atomic<floating-point>, 1205
atomic<integral>, 1205
atomic<shared_ptr<T>>, 576, 577
atomic<T*>, 1205
atomic<weak_ptr<T>>, 578
compare_exchange_weak
atomic, 1205
atomic<floating-point>, 1205
atomic<integral>, 1205
atomic<shared_ptr<T>>, 576, 577
atomic<T*>, 1205
atomic<weak_ptr<T>>, 578
complex
literals, 950
complex, 944
constructor, 945
imag, 946
operator!=, 947
operator**, 950
operator**, 950
operator**, 950
operator**, 950
operator*, 947
operator==, 946
operator+, 946
operator+, 946
operator-, 946
operator-, 946
operator-, 946
operator/, 947
operator/, 947
real, 946
value_type, 944
concat
path, 1122
condition_variable, 1240
constructor, 1240
destructor, 1241
notify_all, 1241
notify_one, 1241
wait, 1241
wait_for, 1242, 1243
wait_until, 1241, 1242
condition_variable_any, 1243
constructor, 1243
destructor, 1244
notify_all, 1244
notify_one, 1244
wait, 1244
wait_for, 1245
wait_until, 1244, 1245
conj, 949
complex, 948
conjunction, 635
const_iterator
basic_string, 669
basic_string_view, 695, 698
const_mem_fun1_ref_t
zombie, 428
const_mem_fun1_t
zombie, 428
const_mem_fun_ref_t
zombie, 428
const_mem_fun_t
zombie, 428
const_ptr
allocator_traits, 548
basic_string, 669
basic_string_view, 695
scoped_allocator_adaptor, 589
const_pointer_cast
shared_ptr, 570
const_reference
basic_string, 669
basic_string_view, 695
const_reverse_iterator
basic_string, 669
basic_string_view, 695
const_void_pointer
allocator_traits, 548
scoped_allocator_adaptor, 589
construct
allocator, 1318
allocator_traits, 549
polymorphic_allocator, 582, 583
scoped_allocator_adaptor, 591–593
consume
memory_order, 1201
converted
wstring_convert, 1325
copy, 912
basic_string, 684
basic_string_view, 699
Index of library names

ctype<char>, 719
classic_table, 721
constructor, 720
cctype<char>, 720
destructor, 720
do_narrow, 721
do_tolower, 721
do_toupper, 721
do_widen, 721
is, 720
narrow, 720
scan_is, 720
scan_not, 720
table, 720
tolower, 720
toupper, 720
widen, 720
cctype_base, 716
do_scan_is, 718
cctype_byname, 719
curr_symbol
money_punct, 744
current_exception, 460
current_path, 1147
cv_status, 1239
cyl_bessel_i, 1027
cyl_bessel_if, 1027
cyl_bessel_il, 1027
cyl_bessel_j, 1028
cyl_bessel_jf, 1028
cyl_bessel_jl, 1028
cyl_bessel_k, 1028
cyl_bessel_kf, 1028
cyl_bessel_kl, 1028
cyl_neumann, 1028
cyl_neumannf, 1028
cyl_neumannl, 1028
data
array, 787
basic_string, 684
basic_string_view, 699
data(T (&array)[N]), 883
data(T &array), 883
data(T&&array), 883
data(C&& c), 883
create_directories, 1145
current_exception, 460
curr_symbol
money_punct, 744
current_exception, 460
current_path, 1147
cv_status, 1239
cyl_bessel_i, 1027
cyl_bessel_if, 1027
cyl_bessel_il, 1027
cyl_bessel_j, 1028
cyl_bessel_jf, 1028
cyl_bessel_jl, 1028
cyl_bessel_k, 1028
cyl_bessel_kf, 1028
cyl_bessel_kl, 1028
cyl_neumann, 1028
cyl_neumannf, 1028
cyl_neumannl, 1028
data
array, 787
basic_string, 684
basic_string_view, 699
data(T (&array)[N]), 883
data(T &array), 883
data(T&&array), 883
data(C&& c), 883
create_directories, 1145
current_path, 1147
create_directory, 1146
current_exception, 460
curr_symbol
money_punct, 744
current_exception, 460
current_path, 1147
create_directory_symlink, 1146
curr_symbol
money_punct, 744
current_exception, 460
create_hard_link, 1146
curr_symbol
money_punct, 744
current_exception, 460
current_path, 1147
current_exception, 460
current_path, 1147
create_directory, 1146
current_exception, 460
curr_symbol
money_punct, 744
current_exception, 460
create_directory_symlink, 1146
current_exception, 460
create_hard_link, 1146
create_directory, 1146
current_exception, 460
curr_symbol
money_punct, 744
current_exception, 460
current_path, 1147
current_exception, 460
current_path, 1147
create_directory_symlink, 1146
create_hard_link, 1146
create_directory, 1146
curr_symbol
money_punct, 744
current_exception, 460
current_path, 1147
create_directory, 1146
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBL_TRUE_MIN</td>
<td>445</td>
</tr>
<tr>
<td>deallocate</td>
<td></td>
</tr>
<tr>
<td>allocator, 549</td>
<td></td>
</tr>
<tr>
<td>allocator_traits, 548</td>
<td></td>
</tr>
<tr>
<td>memory_resource, 580</td>
<td></td>
</tr>
<tr>
<td>polymorphic_allocator, 582</td>
<td></td>
</tr>
<tr>
<td>scoped_allocator_adaptor, 591</td>
<td></td>
</tr>
<tr>
<td>dec, 1051, 1076</td>
<td></td>
</tr>
<tr>
<td>decay, 633</td>
<td></td>
</tr>
<tr>
<td>DECAY_COPY, 1216</td>
<td></td>
</tr>
<tr>
<td>DECIMAL_DIG, 445</td>
<td></td>
</tr>
<tr>
<td>decimal_point</td>
<td></td>
</tr>
<tr>
<td>moneypunct, 744</td>
<td></td>
</tr>
<tr>
<td>numpunct, 732</td>
<td></td>
</tr>
<tr>
<td>declare_no_pointers, 545</td>
<td></td>
</tr>
<tr>
<td>declare_reachable, 545</td>
<td></td>
</tr>
<tr>
<td>decval, 491</td>
<td></td>
</tr>
<tr>
<td>default_delete</td>
<td></td>
</tr>
<tr>
<td>constructor, 553</td>
<td></td>
</tr>
<tr>
<td>operator(), 553, 554</td>
<td></td>
</tr>
<tr>
<td>default_error_condition</td>
<td></td>
</tr>
<tr>
<td>error_category, 480</td>
<td></td>
</tr>
<tr>
<td>error_code, 482</td>
<td></td>
</tr>
<tr>
<td>default_random_engine, 967</td>
<td></td>
</tr>
<tr>
<td>default_searcher, 612</td>
<td></td>
</tr>
<tr>
<td>constructor, 612</td>
<td></td>
</tr>
<tr>
<td>operator(), 612</td>
<td></td>
</tr>
<tr>
<td>defaultfloat, 1051</td>
<td></td>
</tr>
<tr>
<td>defer_lock, 1229</td>
<td></td>
</tr>
<tr>
<td>defer_lock_t, 1229</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td></td>
</tr>
<tr>
<td>operator, 429, 450–453, 552</td>
<td></td>
</tr>
<tr>
<td>denorm_absent, 439</td>
<td></td>
</tr>
<tr>
<td>denorm_indeterminate, 439</td>
<td></td>
</tr>
<tr>
<td>denorm_min</td>
<td></td>
</tr>
<tr>
<td>numeric_limits, 443</td>
<td></td>
</tr>
<tr>
<td>denorm_present, 439</td>
<td></td>
</tr>
<tr>
<td>densities</td>
<td></td>
</tr>
<tr>
<td>piecewise_constant_distribution, 985</td>
<td></td>
</tr>
<tr>
<td>piecewise_linear_distribution, 987</td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td></td>
</tr>
<tr>
<td>recursive_directory_iterator, 1142</td>
<td></td>
</tr>
<tr>
<td>deque, 787</td>
<td></td>
</tr>
<tr>
<td>constructor, 789, 790</td>
<td></td>
</tr>
<tr>
<td>shrink_to_fit, 790</td>
<td></td>
</tr>
<tr>
<td>swap, 791</td>
<td></td>
</tr>
<tr>
<td>destroy, 552</td>
<td></td>
</tr>
<tr>
<td>allocator, 1318</td>
<td></td>
</tr>
<tr>
<td>allocator_traits, 549</td>
<td></td>
</tr>
<tr>
<td>polymorphic_allocator, 583</td>
<td></td>
</tr>
<tr>
<td>scoped_allocator_adaptor, 593</td>
<td></td>
</tr>
<tr>
<td>destroy_at, 552</td>
<td></td>
</tr>
<tr>
<td>destroy_n, 552</td>
<td></td>
</tr>
<tr>
<td>detach</td>
<td></td>
</tr>
<tr>
<td>thread, 1220</td>
<td></td>
</tr>
<tr>
<td>difference_type</td>
<td></td>
</tr>
<tr>
<td>allocator_traits, 548</td>
<td></td>
</tr>
<tr>
<td>basic_string, 669</td>
<td></td>
</tr>
<tr>
<td>basic_string_view, 695</td>
<td></td>
</tr>
<tr>
<td>iterator_traits, 863</td>
<td></td>
</tr>
<tr>
<td>pointer_traits, 544</td>
<td></td>
</tr>
<tr>
<td>scoped_allocator_adaptor, 589</td>
<td></td>
</tr>
<tr>
<td>difftime, 653</td>
<td></td>
</tr>
<tr>
<td>digits</td>
<td></td>
</tr>
<tr>
<td>numeric_limits, 441</td>
<td></td>
</tr>
<tr>
<td>digits10</td>
<td></td>
</tr>
<tr>
<td>directory_entry, 1134</td>
<td></td>
</tr>
<tr>
<td>assign, 1135</td>
<td></td>
</tr>
<tr>
<td>constructor, 1135</td>
<td></td>
</tr>
<tr>
<td>exists, 1136</td>
<td></td>
</tr>
<tr>
<td>file_size, 1137</td>
<td></td>
</tr>
<tr>
<td>hard_link_count, 1137</td>
<td></td>
</tr>
<tr>
<td>is_block_file, 1137</td>
<td></td>
</tr>
<tr>
<td>is_character_file, 1136</td>
<td></td>
</tr>
<tr>
<td>is_directory, 1136</td>
<td></td>
</tr>
<tr>
<td>is_fifo, 1136</td>
<td></td>
</tr>
<tr>
<td>is_other, 1136</td>
<td></td>
</tr>
<tr>
<td>is_regular_file, 1136</td>
<td></td>
</tr>
<tr>
<td>is_socket, 1137</td>
<td></td>
</tr>
<tr>
<td>is_symlink, 1137</td>
<td></td>
</tr>
<tr>
<td>last_write_time, 1137</td>
<td></td>
</tr>
<tr>
<td>operator const filesystem::path&amp;, 1136</td>
<td></td>
</tr>
<tr>
<td>operator!=, 1137</td>
<td></td>
</tr>
<tr>
<td>operator&lt;, 1137</td>
<td></td>
</tr>
<tr>
<td>operator&lt;, 1137</td>
<td></td>
</tr>
<tr>
<td>operator&gt;=, 1138</td>
<td></td>
</tr>
<tr>
<td>operator//=, 1139</td>
<td></td>
</tr>
<tr>
<td>path, 1136</td>
<td></td>
</tr>
<tr>
<td>refresh, 1136</td>
<td></td>
</tr>
<tr>
<td>replace_filename, 1136</td>
<td></td>
</tr>
<tr>
<td>status, 1137</td>
<td></td>
</tr>
<tr>
<td>symlink_status, 1137</td>
<td></td>
</tr>
<tr>
<td>directory_iterator, 1138</td>
<td></td>
</tr>
<tr>
<td>constructor, 1139</td>
<td></td>
</tr>
<tr>
<td>increment, 1139</td>
<td></td>
</tr>
<tr>
<td>operator++, 1139</td>
<td></td>
</tr>
<tr>
<td>operator=, 1139</td>
<td></td>
</tr>
<tr>
<td>directory_options, 1131</td>
<td></td>
</tr>
<tr>
<td>disable_recursion_pending</td>
<td></td>
</tr>
<tr>
<td>recursive_directory_iterator, 1142</td>
<td></td>
</tr>
<tr>
<td>discard_block_engine, 964</td>
<td></td>
</tr>
<tr>
<td>constructor, 964</td>
<td></td>
</tr>
<tr>
<td>discrete_distribution, 983</td>
<td></td>
</tr>
<tr>
<td>constructor, 983</td>
<td></td>
</tr>
<tr>
<td>probabilities, 984</td>
<td></td>
</tr>
<tr>
<td>disjunction, 635</td>
<td></td>
</tr>
<tr>
<td>distance, 865</td>
<td></td>
</tr>
<tr>
<td>div, 435, 1156</td>
<td></td>
</tr>
<tr>
<td>div_t, 435</td>
<td></td>
</tr>
<tr>
<td>divides, 599</td>
<td></td>
</tr>
<tr>
<td>first_argument_type, 1313</td>
<td></td>
</tr>
<tr>
<td>operator(), 599</td>
<td></td>
</tr>
<tr>
<td>result_type, 1313</td>
<td></td>
</tr>
<tr>
<td>second_argument_type, 1313</td>
<td></td>
</tr>
<tr>
<td>divides&lt;, 599</td>
<td></td>
</tr>
<tr>
<td>operator(), 599</td>
<td></td>
</tr>
</tbody>
</table>

Index of library names
do_allocate
    memory_resource, 580
    monotonic_buffer_resource, 588
    synchronized_pool_resource, 586
    unsynchronized_pool_resource, 586

do_allocate_noconv
    codecvt, 724

do_close
    message, 746

do_compare
    collate, 734

do_curr_symbol
    moneypunct, 744

do_date_order
    time_get, 737

do_deallocate
    memory_resource, 580
    monotonic_buffer_resource, 588
    synchronized_pool_resource, 586
    unsynchronized_pool_resource, 586

do_decimal_point
    moneypunct, 744
    numpunct, 733

do_encoding
    codecvt, 724

do_falsename
    numpunct, 733

do_frac_digits
    moneypunct, 744

do_get
    messages, 746
    money_get, 740
    num_get, 726, 728
    time_get, 738

do_get_date
    time_get, 737

do_get_monthname
    time_get, 737

do_get_time
    time_get, 737

do_get_weekday
    time_get, 737

do_get_year
    time_get, 738

do_grouping
    moneypunct, 744
    numpunct, 733

do_hash
    collate, 734

do_in
    codecvt, 723

do_is
    ctype, 717

do_is_equal
    memory_resource, 580
    monotonic_buffer_resource, 588
    synchronized_pool_resource, 586
    unsynchronized_pool_resource, 586

do_length

Index of library names 1396
Index of library names

operator bool, 483
operator! =, 485
operator<, 484
operator<<, 483
operator=, 482
operator==, 484, 485
value, 482

error_condition, 477, 483
assign, 484
category, 484
clear, 484
constructor, 483
message, 484
operator bool, 484
operator! =, 485
operator<, 484
operator=, 484
operator==, 484, 485
value, 484

error_type, 1168, 1169
ESPIPE, 476
ESRCH, 476
ETIME, 476
ETIMEDOUT, 476
ETXTBSY, 476
EWOULDBLOCK, 476

element

exception, 458
constructor, 458
destructor, 458
operator=, 458
what, 458

exception_ptr, 459

exceptions

basic_ios, 1049

exchange, 490

atomic, 1205
atomic<floating-point>, 1205
atomic<integral>, 1205
atomic<shared_ptr<T>>, 576
atomic<T>, 1205
atomic<weak_ptr<T>>, 576

exclusive_scan, 1012

EXDEV, 476

execution

par, 656
par_unseq, 656
seq, 656

execution::parallel_policy, 656

execution::parallel_unsequenced_policy, 656

execution::sequenced_policy, 655

exists, 1148
directory_entry, 1136

exit, 71, 73, 136, 417, 435, 447, 454
EXIT_FAILURE, 435
EXIT_SUCCESS, 435

exp, 1016

complex, 949

valarray, 997

exp2, 1016
exp2f, 1016
exp2l, 1016
expf, 1016
expint, 1029
expintf, 1029
expintl, 1029

expired

weak_ptr, 573

exp1, 1016
exp1f, 1016
exp1l, 1016

exponential_distribution, 975
constructor, 976

lambda, 976

extension

path, 1125
extent, 629

extreme_value_distribution, 978

a, 978
b, 978

constructor, 978

fabs, 1016
fabsf, 1016
fabsl, 1016

locale

fail

basic_ios, 1049

failed

ostreambuf_iterator, 882

false_type, 621

false

numpunct, 733

fclose, 1095, 1154
fdim, 1016
fdimf, 1016
fdiml, 1016

FE_ALL_EXCEPT, 941

FE_DFL_ENV, 941

FE_DIVBYZERO, 941

FE_DOWNWARD, 941

FE_INEXACT, 941

FE_INVALID, 941

FE_OVERFLOW, 941

FE_TONEAREST, 941

FE_TOWARDZERO, 941

FE_UNDERFLOW, 941

FE_UPWARD, 941

feclexcept, 941

fenv_t, 941

fegetenv, 941

fegetexceptflag, 941

fegetround, 941

feholdexcept, 941

fenv_t, 941

feyof, 1154

feraiseexcept, 941

Index of library names
Index of library names

ferror, 1154
fesetenv, 941
fesetexceptflag, 941
fesetround, 941
fetch_add
  atomic<floating-point>, 1209
  atomic<integral>, 1208
  atomic<T*>, 1211
fetch_and
  atomic<integral>, 1208
fetch_or
  atomic<integral>, 1208
fetch_sub
  atomic<floating-point>, 1209
  atomic<integral>, 1208
  atomic<T*>, 1211
fetch_xor
  atomic<integral>, 1208
fetestexcept, 941
feupdateenv, 941
fexcept_t, 941
fflush, 1154
fgetc, 1154
fgetpos, 1154
fgets, 1154
fgetwc, 705
fgetws, 705
FILE, 1154
file_size, 1148
  directory_entry, 1137
file_status, 1131
  constructor, 1133
  permissions, 1133, 1134
  type, 1133, 1134
file_type, 1130
filebuf, 1033, 1091
filename
  path, 1125
FILENAME_MAX, 1154
filesystem_error, 1129
  constructor, 1130
  path1, 1130
  path2, 1130
  what, 1130
fill, 916
  array, 787
  basic_ios, 1047
fill_n, 916
find, 906
  basic_string, 685
  basic_string_view, 701
find_end, 907
find_if_not_of
  basic_string, 687
  basic_string_view, 702
find_last_of
  basic_string, 686
  basic_string_view, 701
first_argument_type
  binary_negate, 1316
  bit_and, 1313
  bit_or, 1313
  bit_xor, 1313
  divides, 1313
  equal_to, 1313
  function, 1313
  greater, 1313
  greater_equal, 1313
  less, 1313
  less_equal, 1313
  logical_and, 1313
  logical_or, 1313
  map::value_compare, 1316
  mem_fn, 1315
  minus, 1313
  modulus, 1313
  multimap::value_compare, 1316
  multiplies, 1313
  not_equal_to, 1313
  owner_less, 1313
  plus, 1313
  reference_wrapper, 1315
fisher_f_distribution, 981
  constructor, 982
  m, 982
  n, 982
fixed, 1051
fixed
  chars_format, 656
flag_type
  basic_regex, 1176
flags
  ios_base, 716, 1042
flip
  bitset, 537
  vector<bool>, 809
float_denorm_style, 438, 439
numeric_limits, 442
float_round_style, 438, 439
float_t, 1016
floor, 1016
  duration, 648
  time_point, 651
floorf, 1016
floorl, 1016
FLT_DECIMAL_DIG, 445
FLT_DIG, 445
FLT_EPSILON, 445
FLT_EVAL_METHOD, 445
FLT_HAS_SUBNORM, 445
FLT_MANT_DIG, 445
FLT_MAX, 445
FLT_MAX_10_EXP, 445
FLT_MAX_EXP, 445
FLT_MIN, 445
FLT_MIN_10_EXP, 445
FLT_MIN_EXP, 445
FLT_RADIX, 445
FLT_ROUNDS, 445
FLT_TRUE_MIN, 445
flush, 1042, 1063, 1074, 1078
  basic_ostream, 1078
fma, 1016
fmaf, 1016
fmal, 1016
fmax, 1016
fmaxf, 1016
fmaxl, 1016
fmin, 1016
fminf, 1016
fminl, 1016
fmod, 1016
fmodf, 1016
fmodl, 1016
fmtflags
  ios_base, 1040, 1079
fopen, 1094, 1154
FOPEN_MAX, 1154
for_each, 905, 906
for_each_n, 906
format
  match_results, 1184, 1185
format_default, 1166, 1168
format_first_only, 1166, 1168, 1189
format_no_copy, 1166, 1168, 1189
format_sed, 1166, 1168
forward, 490
forward_as_tuple, 501
forward_iterator_tag, 864
forward_list
  before_begin, 794
cbegin, 794
clear, 795
constructor, 793, 794
emplace_after, 795
emplace_front, 794
erase_after, 795
erased, 795
front, 794
insert_after, 794, 795
merge, 796
pop, 794
push_front, 794
remove, 796
remove_if, 796
resize, 795
reverse, 797
sort, 797
splice_after, 795, 796
swap, 797
unique, 796
FP_FAST_FMA, 1016
FP_FAST_FMAF, 1016
FP_FAST_FMAL, 1016
FP_ILOGB0, 1016
FP_ILOGBNAN, 1016
FP_INFINITE, 1016
FP_NAN, 1016
FP_NORMAL, 1016
FP_SUBNORMAL, 1016
FP_ZERO, 1016
fpclassify, 1016
fpos, 1037, 1044, 1045
  state, 1045
fpos_t, 1154
fprintf, 1154
fputc, 1154
fputs, 1154
fputwc, 705
fputws, 705
frac_digits
  moneypunct, 744
fread, 1154
free, 435, 552
freeze
  ostrstream, 1311
  strstream, 1312
  strstreambuf, 1307
freopen, 1154
fread, 1154
freadf, 1016
freadpl, 1016
from_bytes
  wstring_convert, 1325
from_chars, 658
from_chars_result, 657
  ec, 657
  ptr, 657
from_time_t
  system_clock, 652
front
  basic_string, 678
  basic_string_view, 699
  forward_list, 794
front_insert_iterator, 870
  constructor, 871
  operator*, 871
  operator++, 871
  operator*, 871
front_inserter, 871
fscanf, 1154
fseek, 1094, 1154
fsetpos, 1154
fstream, 1033, 1091
ftell, 1154
function, 608
  argument_type, 1313
  constructor, 609

Index of library names
Index of library names

deructor, 610
first_argument_type, 1313
invocation, 611
operator bool, 610
operator!=, 611
operator(), 611
operator==, 610
operator==, 611
result_type, 1313
second_argument_type, 1313
swap, 610, 611
target, 611
target_type, 611

future, 1250
constructor, 1251
get, 1251
operator==, 1251
share, 1251
valid, 1252
wait, 1252
wait_for, 1252
wait_until, 1252
future_category, 1246
future_errc, 1245
make_error_code, 1246
make_error_condition, 1246
future_error, 1247
code, 1247
constructor, 1247
what, 1247
fwide, 705
fwprintf, 705
fwrite, 1154
fwscanf, 705

gamma_distribution, 976
alpha, 977
beta, 977
constructor, 977
gbump
basic_streambuf, 1056
gcd, 1016
gcount
basic_istream, 1066
gen
chars_format, 656
GENERALIZED_NONCOMMUTATIVE_SUM, 940
GENERALIZED_SUM, 940
generate, 916
seed_seq, 969
generate_canonical, 970
generate_n, 916
generic_category, 479, 481
generic_string
path, 1124
generic_u16string
path, 1124
generic_u32string
path, 1124
generic_u8string
path, 1124
generic_wstring
path, 1124
geometric_distribution, 973
constructor, 974
p, 974
get
array, 787
basic_istream, 1067
future, 1251
messages, 745
money_get, 740
num_get, 726
pair, 495
reference_wrapper, 598
shared_future, 1254
shared_ptr, 566
time_get, 736
tuple, 503, 504
unique_ptr, 557
variant, 526
get_allocator
basic_string, 685
basic_syncbuf, 1106
match_results, 1185
get_date
time_get, 736
get_default_resource, 584
get_deleter
shared_ptr, 571
unique_ptr, 557
get_future
packaged_task, 1257
promise, 1249
get_id
this_thread, 1220
thread, 1220
get_if, 526
get_money, 1080
get_monthname
time_get, 736
get_new_handler, 429, 454
get_pointer_safety, 546
get_temporary_buffer, 1319
generate_canonical, 970
generate_n, 916
generic_category, 479, 481
generic_string
path, 1124
getenv, 435, 469
getline
  basic_istream, 1067, 1068
basic_string, 692
getloc, 1171
  basic_regex, 1176
basic_streambuf, 1055
ios_base, 1043
getwc, 705
getwchar, 705
locale, 715
gmtime, 653
good
  basic_ios, 1049
gptr
  basic_streambuf, 1056
greater, 601
  first_argument_type, 1313
operator(), 601
partial_ordering, 464
result_type, 1313
second_argument_type, 1313
strong_ordering, 467
weak_ordering, 465
greater<, 601
operator(), 601
greater_equal, 601
  first_argument_type, 1313
operator(), 602
result_type, 1313
second_argument_type, 1313
greater_equal<, 602
operator(), 602
grouping
  moneypunct, 744
numpunct, 733
gslice, 999
  constructor, 1001
size, 1001
start, 1001
stride, 1001
gslice_array, 1001
  operator*=, 1002
operator+=, 1002
operator-=, 1002
operator/=, 1002
operator<<=, 1002
operator=, 1001, 1002
operator>>, 1002
operator%=, 1002
operator%==, 1002
operator&=, 1002
operator^=, 1002
operator|=, 1002
value_type, 1001
hardware_concurrency
  thread, 1220
hardware_constructive_interference_size, 455
hardware-destructive_interference_size, 455
has_denorm_loss
  numeric_limits, 442
has_extension
  path, 1126
has_facet
  locale, 715
has_filename
  path, 1126
has_infinity
  numeric_limits, 442
has_parent_path
  path, 1126
has_quiet_NaN
  numeric_limits, 442
has_relative_path
  path, 1126
has_root_directory
  path, 1126
has_root_name
  path, 1126
has_root_path
  path, 1126
has_signaling_NaN
  numeric_limits, 442
has_stem
  path, 1126
has_unique_object_representations, 627, 628
has_value
  any, 532
  optional, 514
has_virtual_destructor, 627
hash, 614
  argument_type, 1315
  collate, 734
  error_code, 485
  monostate, 528
  optional, 517
  pmr::string, 694
  pmr::u16string, 694
  pmr::u32string, 694
  pmr::wstring, 694
  result_type, 1315
  shared_ptr, 575
  string, 694
  string_view, 703
  thread::id, 1218
  type_index, 654
  u16string, 694
  u16string_view, 703
  u32string, 694
  u32string_view, 703
  unique_ptr, 575
  variant, 528
  wstring, 694
  wstring_view, 703

Index of library names 1403
hash_code, 538
  type_index, 654
  type_info, 456
hash_value
  path, 1128
hermite, 1029
hermitef, 1029
hermitel, 1029
hex, 1051
hex
  chars_format, 656
hexfloat, 1051
high_resolution_clock, 653
holds_alternative, 526
HUGE_VAL, 1016
HUGE_VALF, 1016
HUGE_VALL, 1016
hypot, 1016
  3-argument form, 1025
hypotf, 1016
hypotl, 1016
id
  locale, 713
ifstream, 1033, 1091
ignore, 502
  basic_istream, 1068
ilogb, 1016
ilogbf, 1016
ilogbl, 1016
imag, 949
complex, 946, 948
imaxabs, 1156
imaxdiv, 1156
imaxdiv_t, 1156
imbue, 1171
  basic_filebuf, 1097
  basic_ios, 1047
  basic_regex, 1176
  basic_streambuf, 1057
  ios_base, 1043
in
  codecvt, 722
  in_avail
    basic_streambuf, 1055
includes, 929
inclusive_scan, 1012
increment
  directory_iterator, 1139
  recursive_directory_iterator, 1142
independent_bits_engine, 964, 965
index
  variant, 525
index_sequence, 488
index_sequence_for, 488
indirect_array, 1003
operator++, 1004
operator+=, 1004
operator-=, 1004
operator/=, 1004
operator<<=, 1004
operator=, 1004
operator>>=, 1004
operator[], 1003
operator%==, 1004
operator&=, 1004
operator*=, 1004
operator|=, 1004
value_type, 1003
INFINITY, 1016
infinity
  numeric_limits, 442
init
  basic_ios, 1047, 1062
  basic_ostream, 1073
initializer_list, 461
  begin, 462
  constructor, 461
  end, 462
  size, 462
inner_alloc<br>inner_allocator
  scoped_allocator_adaptor, 591
inner_allocator_type
  scoped_allocator_adaptor, 590
inner_product, 1009
inplace_merge, 929
input_iterator_merge, 864
insert
  basic_string, 681, 682
dque, 790
list, 800
map, 815
multimap, 820
unordered_map, 833
unordered_multimap, 839
vector, 806
insert_after
  forward_list, 794, 795
insert_iterator, 871
  constructor, 872
  operator*, 872
  operator++, 872
  operator=, 872
insert_or_assign
  map, 816
unordered_map, 834
inserter, 872
int16_t, 446
int32_t, 446
int64_t, 446
int8_t, 446
int_fast16_t, 446
int_fast32_t, 446
int_fast64_t, 446
int_fast8_t, 446
int_least16_t, 446
int_least32_t, 446
int_least64_t, 446
int_least8_t, 446
int_least8_t, 446
INT_MAX, 445
INT_MIN, 445

int_type
  - char_traits, 662
    - wstring_convert, 1325
integer_sequence, 491
  - value_type, 491
integral_constant, 621
  - value_type, 621
internal, 1050

intervals
  - piecewise_constant_distribution, 985
    - piecewise_linear_distribution, 987
intmax_t, 446
intptr_t, 446
invalid_argument, 472, 473, 535, 536
  - constructor, 473
INVOKC, 596, 597
invoke, 597
io_errc, 1037
make_error_code, 1051
make_error_condition, 1051
ios, 1033, 1037
ios_base, 1038
  - constructor, 1044
    - destructor, 1044
failure, 1040
flags, 1042
fmtflags, 1040
getloc, 1043
imbue, 1043
Init, 1042
iostate, 1040
ivord, 1043
openmode, 1040
precision, 1042
pword, 1044
register_callback, 1044
seekdir, 1040
setf, 1042
sync_with_stdio, 1043
unsetf, 1042
width, 1042
xalloc, 1043
ios_base::failure, 1040
  - constructor, 1040
ios_base::Init, 1042
  - constructor, 1042
    - destructor, 1042
iostate
  - ios_base, 1040
iostream_category, 1051
iota, 1016
is
  - ctype, 717
    - ctype<
      - char>, 720
is_absolute
  - path, 1126

is_abstract, 623
is_aggregate, 623
is_always_equal
  - allocator, 549
    - allocator_traits, 548
      - scoped_allocator_adaptor, 590
is_always_lock_free
  - atomic, 1205
    - atomic<
      - floating-point>, 1205
      - atomic<integral>, 1205
      - atomic<T*>, 1205
is_arithmetic, 622
is_array, 622
is_assignable, 624
is_base_of, 629
is_bind_expression, 606
is_block_file, 1148
  - directory_entry, 1136
is_bounded
  - numeric_limits, 443
is_character_file, 1148
  - directory_entry, 1136
is_class, 622
is_compound, 622
is_const, 623
is_constructible, 624, 628
is_convertible, 629, 630
is_copy_assignable, 625
is_copy_constructible, 624
is_default_constructible, 624
is_destructible, 625
is_directory, 1149
  - directory_entry, 1136
is_empty
  - class, 623
    - function, 1149
is_enum, 622
is_eq, 462
is_equal
  - memory_resource, 580
is_error_code_enum, 477
is_error_condition_enum, 477
is_exact
  - numeric_limits, 441
is_execution_policy, 655
is_fifo, 1149
  - directory_entry, 1136
is_final, 623
is_floating_point, 622
is_function, 622
is_fundamental, 622
is_geq, 462
is_gt, 462
is_gteq, 462
is_heap, 933, 934
is_heap_until, 934
is_iec559
  - numeric_limits, 443
is_integer
numeric_limits, 441
is_integral, 622
is_invocable, 629
is_invocable_r, 630
is_literal_type, 1319
is_lock_free
  atomic, 1205
  atomic<floating-point>, 1205
  atomic<integral>, 1205
  atomic<T*>, 1205
is_lt, 462
is_lteq, 462
is_lvalue_reference, 622
is_member_function_pointer, 622
is_member_object_pointer, 622
is_member_pointer, 623
is_modulo
  numeric_limits, 443
is_move_assignable, 625
is_move_constructible, 624
is_neq, 462
is_nothrow_assignable, 627
is_nothrow_constructible, 626
is_nothrow_copy_assignable, 627
is_nothrow_copy_constructible, 627
is_nothrow_default_constructible, 626
is_nothrow_destructible, 626
is_nothrow_invocable, 630
is_nothrow_invocable_r, 630
is_nothrow_move_assignable, 627
is_nothrow_move_constructible, 627
is_nothrow_swappable, 627
is_nothrow_swappable_with, 627
is_null_pointer, 622
is_object, 622
is_open
  basic_filebuf, 1094
  basic_fstream, 1103
  basic_ifstream, 1099
  basic_ofstream, 1101
is_other, 1149
  directory_entry, 1136
is_partitioned, 926
is_permutation, 910
is_placeholder, 606
is_pointer, 622
is_polymorphic, 623
is_reference, 622
is_regular_file, 1149, 1150
  directory_entry, 1136
is_relative
  path, 1126
is_rvalue_reference, 622
is_scalar, 622
is_signed
  class, 624
  numeric_limits, 441
is_socket, 1150
  directory_entry, 1137
is_sorted, 924
is_sorted_until, 924
is_standard_layout, 623
is_swappable, 625
is_swappable_with, 625
is_symlink, 1150
  directory_entry, 1137
is_trivial, 623
is_trivially_assignable, 626
is_trivially_constructible, 626
is_trivially_copy_assignable, 626
is_trivially_copy_constructible, 626
is_trivially_copyable, 623
is_trivially_default_constructible, 626
is_trivially_destructible, 626
is_trivially_move_assignable, 626
is_trivially_move_constructible, 626
is_union, 622
is_unsigned, 624
is_void, 622
is_volatile, 622
is_volatile<
  floating-point>, 1205
  integral>, 1205
  T*>, 1205
isalpha, 704, 715
isalnum, 704, 715
iscntrl, 704, 715
isdigit, 704, 715
isfinite, 1016
isgraph, 704, 715
isgreater, 1016
isgreaterequal, 1016
isinf, 1016
isless, 1016
islessequal, 1016
islessgreater, 1016
isnan, 1016
isnormal, 1016
isprint, 704, 715
ispunct, 704, 715
isspace, 704, 715
istream, 1033, 1060
istream_iterator, 876
  constructor, 877
destructor, 877
operator!=, 878
operator*, 878
operator++, 878
operator->, 878
operator==, 878
istreambuf_iterator, 879
  constructor, 880, 881
equal, 881
operator!=, 881
operator*, 881
operator++, 881
operator==, 881
regular_expression_traits, 1196
regex_traits, 1170
<table>
<thead>
<tr>
<th>Function/Concept</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>istringstream</code></td>
<td>1033, 1082</td>
</tr>
<tr>
<td><code>istrstream</code></td>
<td>1309</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>1310</td>
</tr>
<tr>
<td><code>rdbuf</code></td>
<td>1310</td>
</tr>
<tr>
<td><code>str</code></td>
<td>1310</td>
</tr>
<tr>
<td><code>isunordered</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>isupper</code></td>
<td>704, 715</td>
</tr>
<tr>
<td><code>iswalnum</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswalpha</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswblank</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswcntrl</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswctype</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswdigit</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswgraph</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswlower</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswprint</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswpunct</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswspace</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswupper</code></td>
<td>704</td>
</tr>
<tr>
<td><code>iswxdigit</code></td>
<td>704</td>
</tr>
<tr>
<td><code>isxdigit</code></td>
<td>704, 715</td>
</tr>
<tr>
<td><code>iter_swap</code></td>
<td>914</td>
</tr>
<tr>
<td><code>iterator</code></td>
<td>1312, 1320</td>
</tr>
<tr>
<td><code>basic_string</code></td>
<td>669</td>
</tr>
<tr>
<td><code>basic_string_view</code></td>
<td>695</td>
</tr>
<tr>
<td><code>iterator_category</code></td>
<td>863</td>
</tr>
<tr>
<td><code>iterator_traits</code></td>
<td>863</td>
</tr>
<tr>
<td><code>difference_type</code></td>
<td>863</td>
</tr>
<tr>
<td><code>iterator_category</code></td>
<td>863</td>
</tr>
<tr>
<td><code>pointer</code></td>
<td>863</td>
</tr>
<tr>
<td><code>reference</code></td>
<td>863</td>
</tr>
<tr>
<td><code>value_type</code></td>
<td>863</td>
</tr>
<tr>
<td><code>iword</code></td>
<td>1043</td>
</tr>
<tr>
<td><code>ios_base</code></td>
<td>1043</td>
</tr>
<tr>
<td><code>jmp_buf</code></td>
<td>470</td>
</tr>
<tr>
<td><code>join</code></td>
<td>1220</td>
</tr>
<tr>
<td><code>joinable</code></td>
<td>1219</td>
</tr>
<tr>
<td><code>k</code></td>
<td>975</td>
</tr>
<tr>
<td><code>negative_binomial_distribution</code></td>
<td>975</td>
</tr>
<tr>
<td><code>kill_dependency</code></td>
<td>1203</td>
</tr>
<tr>
<td><code>knuth_b</code></td>
<td>967</td>
</tr>
<tr>
<td><code>L_tmpnam</code></td>
<td>1154</td>
</tr>
<tr>
<td><code>labs</code></td>
<td>435</td>
</tr>
<tr>
<td><code>laguerre</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>laguerref</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>laguerrel</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>lambda</code></td>
<td>601</td>
</tr>
<tr>
<td><code>exponential_distribution</code></td>
<td>976</td>
</tr>
<tr>
<td><code>last_write_time</code></td>
<td>1150</td>
</tr>
<tr>
<td><code>directory_entry</code></td>
<td>1137</td>
</tr>
<tr>
<td><code>launder</code></td>
<td>454</td>
</tr>
<tr>
<td><code>LC_ALL</code></td>
<td>748</td>
</tr>
<tr>
<td><code>LC_COLLATE</code></td>
<td>748</td>
</tr>
<tr>
<td><code>LC_CTYPE</code></td>
<td>748</td>
</tr>
<tr>
<td><code>LC_MONETARY</code></td>
<td>748</td>
</tr>
<tr>
<td><code>LC_NUMERIC</code></td>
<td>748</td>
</tr>
<tr>
<td><code>LC_TIME</code></td>
<td>748</td>
</tr>
<tr>
<td><code>lcm</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>lconv</code></td>
<td>748</td>
</tr>
<tr>
<td><code>LDBL_DECIMAL_DIG</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_DIG</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_EPSILON</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_HAS_SUBNORM</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_MANT_DIG</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_MAX</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_MAX_10_EXP</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_MAX_EXP</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_MIN</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_MIN_10_EXP</code></td>
<td>445</td>
</tr>
<tr>
<td><code>LDBL_TRUE_MIN</code></td>
<td>445</td>
</tr>
<tr>
<td><code>ldexp</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>ldexpf</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>ldexpl</code></td>
<td>1016</td>
</tr>
<tr>
<td><code>ldiv</code></td>
<td>435</td>
</tr>
<tr>
<td><code>ldiv_t</code></td>
<td>435</td>
</tr>
<tr>
<td><code>left</code></td>
<td>1050</td>
</tr>
<tr>
<td><code>legendre</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>legendref</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>legendrel</code></td>
<td>1030</td>
</tr>
<tr>
<td><code>length</code></td>
<td>677</td>
</tr>
<tr>
<td><code>basic_string</code></td>
<td>698</td>
</tr>
<tr>
<td><code>basic_string_view</code></td>
<td>698</td>
</tr>
<tr>
<td><code>char_traits</code></td>
<td>676</td>
</tr>
<tr>
<td><code>codecvt</code></td>
<td>722</td>
</tr>
<tr>
<td><code>match_results</code></td>
<td>1183</td>
</tr>
<tr>
<td><code>regex_traits</code></td>
<td>1170</td>
</tr>
<tr>
<td><code>sub_match</code></td>
<td>1177</td>
</tr>
<tr>
<td><code>length_error</code></td>
<td>472, 473, 669</td>
</tr>
<tr>
<td><code>constructor</code></td>
<td>473, 474</td>
</tr>
<tr>
<td><code>less</code></td>
<td>601</td>
</tr>
<tr>
<td><code>first_argument_type</code></td>
<td>1313</td>
</tr>
<tr>
<td><code>operator()</code></td>
<td>601</td>
</tr>
<tr>
<td><code>partial_ordering</code></td>
<td>464</td>
</tr>
<tr>
<td><code>result_type</code></td>
<td>1313</td>
</tr>
<tr>
<td><code>second_argument_type</code></td>
<td>1313</td>
</tr>
<tr>
<td><code>strong_ordering</code></td>
<td>467</td>
</tr>
<tr>
<td><code>weak_ordering</code></td>
<td>465</td>
</tr>
<tr>
<td><code>less&lt;</code></td>
<td>601</td>
</tr>
<tr>
<td><code>operator()</code></td>
<td>601</td>
</tr>
<tr>
<td><code>less_equal</code></td>
<td>602</td>
</tr>
<tr>
<td><code>first_argument_type</code></td>
<td>1313</td>
</tr>
<tr>
<td><code>operator()</code></td>
<td>602</td>
</tr>
<tr>
<td><code>result_type</code></td>
<td>1313</td>
</tr>
<tr>
<td><code>second_argument_type</code></td>
<td>1313</td>
</tr>
<tr>
<td><code>less_equal&lt;</code></td>
<td>602</td>
</tr>
<tr>
<td><code>operator()</code></td>
<td>602</td>
</tr>
<tr>
<td><code>lexically_normal</code></td>
<td>1126</td>
</tr>
<tr>
<td><code>path</code></td>
<td>1127</td>
</tr>
<tr>
<td><code>lexically_proximate</code></td>
<td>1127</td>
</tr>
<tr>
<td><code>lexically_relative</code></td>
<td>1127</td>
</tr>
</tbody>
</table>
Index of library names

path, 1127
lexicographical_compare, 936
lexicographical_compare_3way, 937, 938
lgamma, 1016
lgammaf, 1016
lgammal, 1016
linear_congruential_engine, 960
constructor, 961
list, 797
constructor, 799, 800
splice, 801
swap, 803
literals
complex, 950
little
endian, 636
llabs, 435
lldiv, 435
lldiv_t, 435
LONGLONG_MAX, 445
LONGLONG_MIN, 445
llrint, 1016
llrintf, 1016
llrintl, 1016
llround, 1016
llroundf, 1016
llroundl, 1016
load
atomic, 1205
atomic<floating-point>, 1205
atomic<integer>, 1205
atomic<shared_ptr<T>>, 576
atomic<T>, 1205
atomic<weak_ptr<T>>, 578
locale, 1171, 1176, 1195
category, 711
classic, 715
combine, 714
constructor, 713, 714
destructor, 714
facet, 712
global, 715
has_facet, 715
id, 713
name, 714
operator!=, 714
operator(), 714
operator=, 714
operator==, 714
use_facet, 715
localeconv, 748
localtime, 653
lock, 1238
shared_lock, 1236
unique_lock, 1233
weak_ptr, 573
lock_guard, 1229
constructor, 1229, 1230
destructor, 1230
log, 1016
complex, 949
valarray, 997
log10, 1016
complex, 949
valarray, 997
log10f, 1016
log10l, 1016
log1pf, 1016
log1pl, 1016
log2, 1016
log2f, 1016
log2l, 1016
logb, 1016
logbf, 1016
logbl, 1016
logf, 1016
logic_error, 472
constructor, 472, 473
logical_and, 602
first_argument_type, 1313
operator(), 602
result_type, 1313
second_argument_type, 1313
logical_and<>, 602
operator(), 602
logical_not, 603
argument_type, 1313
operator(), 603
result_type, 1313
logical_not<>, 603
operator(), 603
logical_or, 603
first_argument_type, 1313
operator(), 603
result_type, 1313
second_argument_type, 1313
logical_or<>, 603
operator(), 603
log1l, 1016
lognormal_distribution, 979
constructor, 980
m, 980
s, 980
LONG_MAX, 445
LONG_MIN, 445
longjmp, 470
lookup_classname
regex_traits, 1170
regular expression traits, 1196
lookup_collatename
regex_traits, 1170
regular expression traits, 1196
lower_bound, 925
lowest
numeric_limits, 440
lrint, 1016
lrintf, 1016

© ISO/IEC N4713
lrintl, 1016
lround, 1016
lroundf, 1016
lroundl, 1016

= fisher_f_distribution, 982
lognormal_distribution, 980
make_any, 532
make_error_code
  errc, 483
  future_errc, 1246
  io_errc, 1051
make_error_condition
  errc, 484
  future_errc, 1246
  io_errc, 1051
make_exception_ptr, 460
make_from_tuple, 502
make_heap, 933
make_index_sequence, 488
make_integer_sequence, 491
make_move_iterator, 876
make_optional, 517
make_pair, 494
make_preferred
  path, 1122
make_ready_at_thread_exit
  packaged_task, 1258
make_reverse_iterator, 869
make_shared, 567–569
make_tuple, 501
make_unique, 560
make_unsigned, 632
malloc, 435, 552, 1302
map, 812
  constructor, 815
  insert, 815
  insert_or_assign, 816
  operator<, 815
  operator==, 815
  swap, 817
  try_emplace, 816
map::value_compare
  first_argument_type, 1316
  result_type, 1316
  second_argument_type, 1316
mark_count
  basic_regex, 1176
mask_array, 1002
  operator%:, 1003
  operator%#, 1003
  operator~%, 1003
  operator|%, 1003
  value_type, 1002
match_any, 1166, 1167
match_continuous, 1166, 1167, 1191
match_default, 1166
match_flag_type, 1166, 1167, 1196
match_not_bol, 1166, 1167
match_not_bow, 1166, 1167
match_not_eol, 1166, 1167
match_not_eow, 1166, 1167
match_not_null, 1166, 1167, 1191
match_prev_avail, 1166, 1167, 1191
match_results, 1181, 1190, 1192
  begin, 1184
  constructor, 1182
  empty, 1183
  end, 1184
  format, 1184, 1185
  get_allocator, 1185
  length, 1183
  matched, 1181
  max_size, 1183
  operator! =, 1185
  operator=, 1183
  operator==, 1185
  operator[] , 1184
  position, 1183
  prefix, 1184
  ready, 1183
  size, 1183
  str, 1183
  suffix, 1184
  swap, 1185
MATH_ERREXCEPT, 1016
math_errhandling, 1016
MATH_ERRNO, 1016
max, 935
  duration, 646
  duration_values, 643
  numeric_limits, 440
  time_point, 650
  valarray, 995
max_align_t, 434, 437
max_digits10
  numeric_limits, 441
max_element, 936
max_exponent
  numeric_limits, 442
max_exponent10
  numeric_limits, 442
max_length
codevt, 722
max_size
  allocator, 1318
c_allocator_traits, 549
  array, 785

Index of library names 1409
basic_string, 677
basic_string_view, 698
match_results, 1183
scoped_allocator_adaptor, 591
MB_CUR_MAX, 435
MB_LEN_MAX, 445
mblen, 435, 707
mbrlen, 435, 707
mbrtoc16, 706
mbrtoc32, 706
mbrtoc, 705, 707
mbstowcs, 705
mbsinit, 705, 707
memchr, 704
memcmp, 704
memcpy, 704
memmove, 704
memset, 704
merge, 928
forward_list, 796
list, 802
mersenne_twister_engine, 961
constructor, 962
message
do_close, 746
error_category, 480
error_code, 482
error_condition, 484
messages, 745
close, 745
do_get, 746
do_open, 745
get, 745
open, 745
messages_byname, 746
min, 934
duration, 646
duration_values, 643
numeric_limits, 440
time_point, 650
valarray, 995
min_element, 935
min_exponent
numeric_limits, 441
min_exponent10
numeric_limits, 441
minmax
935
minmax_element
936
minstd_rand, 966
minstd_rand0, 966
minus, 598
first_argument_type, 1313
operator(), 598
result_type, 1313
second_argument_type, 1313
minus, 599
operator(), 599
mismatch, 909
mktime, 653
modf, 1016
modff, 1016
modfl, 1016
modulus, 599
first_argument_type, 1313
operator(), 599
result_type, 1313
second_argument_type, 1313
modulus, 599
operator(), 600

Index of library names
Index of library names

money_get, 740
  do_get, 740
get, 740
money_put, 741
  do_put, 742
put, 742
moneypunct, 742
  curr_symbol, 744
decimal_point, 744
do_curr_symbol, 744
do_decimal_point, 744
do_frac_digits, 744
do_grouping, 744
do_neg_format, 744
do_negative_sign, 744
do_pos_format, 744
do_positive_sign, 744
do_thousands_sep, 744
frac_digits, 744
grouping, 744
negative_sign, 744
positive_sign, 744
thousands_sep, 744
moneypunct_byname, 744
monostate, 527
monotonic_buffer_resource, 587
  constructor, 587, 588
destructor, 588
do_allocate, 588
do_deallocate, 588
do_is_equal, 588
release, 588
upstream_resource, 588
move
  algorithm, 913, 914
basic_ios, 1048
function, 490
move_backward, 914
move_if_noexcept, 491
move_iterator, 873
  base, 874
  constructor, 874
  operator!=, 875
  operator*, 874
  operator+, 875, 876
  operator++, 875
  operator==, 875
  operator-, 875, 876
  operator~, 875
  operator->, 874
  operator--, 875
  operator<, 876
  operator<=, 876
  operator=, 874
  operator==, 875
  operator>, 876
  operator>=, 876
  operator[] , 875
mt19937, 966
  mt19937_64, 967
multimap, 817
  constructor, 820
  insert, 820
  operator<, 820
  operator==, 820
  swap, 820
multimap::value_compare
  first_argument_type, 1316
  result_type, 1316
  second_argument_type, 1316
multiplies, 599
  first_argument_type, 1313
  operator(), 599
  result_type, 1313
  second_argument_type, 1313
multiplies<> , 599
  operator(), 599
multiset, 823
  constructor, 826
  operator<, 826
  operator==, 826
  swap, 826
mutex, 1223
  shared_lock, 1237
  unique_lock, 1234
n
  chi_squared_distribution, 980
  fisher_f_distribution, 982
name
  error_category, 480
  locale, 714
  type_index, 654
  type_info, 456
NAN, 1016
nan, 1016
nanf, 1016
nanl, 1016
narrow
  basic_ios, 1047
  ctype, 717
  ctype<char>, 720
native
  endian, 636
  path, 1123
NDEBUG, 417
nearbyint, 1016
nearbyintf, 1016
nearbyintl, 1016
negate, 600
  argument_type, 1313
  operator(), 600
  result_type, 1313
negate< , 600
  operator(), 600
negation, 636
negative_binomial_distribution, 974
  constructor, 974
Index of library names

k, 975
p, 975
negative_sign
  moneypunct, 744
nested_exception, 460
  constructor, 460
  nested_ptr, 461
  rethrow_if_nested, 461
  rethrow_nested, 461
  throw_with_nested, 461
nested_ptr
  nested_exception, 461
new
  operator, 429, 449–451, 453, 552
new_delete_resource, 583
new_handler, 454
next, 865
next_permutation, 938
nextafter, 1016
nextafterf, 1016
nextafterl, 1016
nexttoward, 1016
nexttowardf, 1016
nexttowardl, 1016
noboolalpha, 1049
none
  bitset, 538
  none_of, 905
nonequal
  strong_equality, 463
nonequivalent
  strong_equality, 463
  weak_equality, 463
norm, 949
  complex, 948
normal_distribution, 978
  constructor, 979
  mean, 979
  stddev, 979
noshowbase, 1050
noshowpoint, 1050
noshowpos, 1050
noskipws, 1050
not1, 1316
not2, 1316, 1317
not_equal_to, 601
  first_argument_type, 1313
  operator(), 601
  result_type, 1313
  second_argument_type, 1313
not_equal_to>, 601
  operator(), 601
not_fn, 605
notthrow, 448
notthrow_t, 448
notify_all
  condition_variable, 1241
  condition_variable_any, 1244
  notify_all_at_thread_exit, 1239

notify_one
  condition_variable, 1241
  condition_variable_any, 1244
nounitbuf, 1050
nouppercase, 1050
nth_element, 921
NULL, 434–436, 653, 705, 748, 1154
null_memory_resource, 583
nullopt, 515
nullopt_t, 515
nullptr_t, 434, 436
num_get, 725
  do_get, 726, 728
  get, 726
num_put, 729
  do_put, 729, 731
  put, 729
numeric_limits, 438, 439
  denorm_min, 443
  digits, 441
  digits10, 441
  epsilon, 441
  float_denorm_style, 442
  has_denorm_loss, 442
  has_infinity, 442
  has_quiet_NaN, 442
  has_signaling_NaN, 442
  infinity, 442
  is_bounded, 443
  is_exact, 441
  is_iec559, 443
  is_integer, 441
  is_modulo, 443
  is_signed, 441
  lowest, 440
  max, 440
  max_digits10, 441
  max_exponent, 442
  max_exponent10, 442
  min, 440
  min_exponent, 441
  min_exponent10, 441
  quiet_NaN, 442
  radix, 441
  round_error, 441
  round_style, 443
  signaling_NaN, 443
  tinyness_before, 443
  traps, 443
numeric_limits<bool>, 444
numpunct, 732
  decimal_point, 732
  do_decimal_point, 733
  do_falsename, 733
  do_grouping, 733
  do_thousands_sep, 733
  do_truename, 733
  falsename, 733
  grouping, 733
thousands_sep, 732
truename, 733
numpunct_byname, 733

c, 1051
off_type
char_traits, 662
offsetof, 434, 436, 1302
ofstream, 1033, 1091
once_flag, 1238
open
basic_filebuf, 1094
basic_fstream, 1103, 1104
basic_ifstream, 1099
basic_ofstream, 1101
messages, 745
openmode
os_base, 1040
operator floating-point
atomic<floating-point>, 1205
operator integral
atomic<integral>, 1205
operator type
atomic, 1205
operator basic_string
sub_match, 1177
operator basic_string_view
basic_string, 685
operator bool
basic_ios, 1049
basic_istream::sentry, 1064
basic_ostream::sentry, 1074
error_code, 483
error_condition, 484
function, 610
optional, 514
shared_lock, 1237
shared_ptr, 566
unique_lock, 1234
unique_ptr, 557
operator const filesystem::path&
directory_entry, 1136
operator partial_ordering
strong_ordering, 467
weak_ordering, 466
operator weak_ptr<T>*
atomic<weak_ptr<T>*>, 576
operator string_type
path, 1123
operator strong_equality
strong_ordering, 467
operator T*
atomic<T*>, 1205
operator T&
reference_wrapper, 598
operator weak_equality
partial_ordering, 465
strong_equality, 464
strong_ordering, 467
weak_ordering, 466
operator weak_ordering
strong_ordering, 467
operator weak_ptr<T>*
atomic<weak_ptr<T>*>, 578
operator!
basic_ios, 1049
valarray, 994
operator!=, 415, 1304
allocator, 550
basic_string, 690
basic_string_view, 703
bitset, 538
complex, 947
directory_entry, 1137
duration, 647
e-error_category, 480
e-error_code, 485
e-error_condition, 485
function, 611
istream_iterator, 878
istreambuf_iterator, 881
locale, 714
match_results, 1185
memory_resource, 580
monostate, 528
move_iterator, 875
optional, 515, 516
pair, 494
partial_ordering, 465
path, 1128
polymorphic_allocator, 583
queue, 850
regex_iterator, 1191
regex_token_iterator, 1194
reverse_iterator, 868
scoped_allocator_adaptor, 593
shared_ptr, 569
stack, 854
strong_equality, 464
strong_ordering, 468
sub_match, 1177–1180
thread::id, 1218
time_point, 651
tuple, 505
type_index, 654
type_info, 456
unique_ptr, 560, 561
valarray, 997
variant, 527
weak_equality, 463
weak_ordering, 466
operator""h
duration, 648
operator""i
complex, 950
operator""if
complex, 950
operator""il

Index of library names
complex, 950
operator""min
duration, 649
operator""ms
duration, 649
operator""ns
duration, 649
operator""s
duration, 649
string, 694
u16string, 694
u32string, 694
wstring, 694
operator""sv
string_view, 703
u16string_view, 703
u32string_view, 703
wstring_view, 703
operator""us
duration, 649
operator()
    binary_negate, 1316
    bit_and, 603
    bit_and<>, 603
    bit_not, 604
    bit_not<>; 604
    bit_or, 604
    bit_or<>; 604
    bit_xor, 604
    bit_xor<>; 604
    boyer_moore_horspool_searcher, 614
    boyer_moore_searcher, 613
    default_delete, 553, 554
    default_searcher, 612
    divides, 599
    divides<>; 599
    equal_to, 600
    equal_to<>; 600
    function, 611
greater, 601
greater<>; 601
greater_equal, 602
greater_equal<>; 602
    less, 601
    less<>; 601
    less_equal, 602
    less_equal<>; 602
    locale, 714
    logical_and, 602
    logical_and<>; 602
    logical_not, 603
    logical_not<>; 603
    logical_or, 603
    logical_or<>; 603
    minus, 598
    minus<>; 599
    modulus, 599
    modulus<>; 600
    multiplies, 599
multiplies<>; 599
negate, 600
negate<>; 600
not_equal_to, 601
not_equal_to<>; 601
owner_less, 574
packaged_task, 1258
plus, 598
plus<>; 598
random_device, 968
reference_wrapper, 598
unary_negate, 1316
operator*
    back_insert_iterator, 870
    complex, 947
duration, 646
    front_insert_iterator, 871
    insert_iterator, 872
    istream_iterator, 878
    istreambuf_iterator, 881
    move_iterator, 874
    optional, 514
    ostream_iterator, 879
    ostreambuf_iterator, 882
    raw_storage_iterator, 1318
    regex_iterator, 1191
    regex_token_iterator, 1194
    reverse_iterator, 867
    shared_ptr, 566
    unique_ptr, 557
    valarray, 996
operator==
    basic_string, 688–690
    complex, 946
duration, 645
gslice_array, 1002
indirect_array, 1004
mask_array, 1003
slice_array, 999
valarray, 994, 995
operator+
    basic_string, 688–690
    complex, 946
duration, 645, 650
    move_iterator, 875, 876
    reverse_iterator, 868, 869
time_point, 650
    valarray, 994, 996
operator++
    atomic<integral>, 1211
    atomic<T*>, 1211
    back_insert_iterator, 870
directory_iterator, 1139
duration, 645
    front_insert_iterator, 871
    insert_iterator, 872
    istream_iterator, 878
    istreambuf_iterator, 881
    move_iterator, 875
    ostream_iterator, 879
Index of library names

operator==
  atomic<floating-point>, 1209
  atomic<integral>, 1208
  atomic<T*>, 1208, 1209, 1211
  basic_string, 678, 679
  complex, 946
  duration, 645
gslice_array, 1002
  indirect_array, 1004
  mask_array, 1003
  move_iterator, 875
  path, 1122
  reverse_iterator, 868
  slice_array, 999
time_point, 650
  valarray, 994, 995

operator--
  complex, 947
  duration, 645, 650
  move_iterator, 875, 876
  reverse_iterator, 868
  time_point, 650
  valarray, 994, 996

operator==
  atomic<floating-point>, 1209
  atomic<integral>, 1208
  atomic<T*>, 1208, 1209, 1211
  complex, 946
  duration, 645
gslice_array, 1002
  indirect_array, 1004
  mask_array, 1003
  move_iterator, 875
  reverse_iterator, 868
  slice_array, 999
time_point, 650
  valarray, 994, 995

operator>
  basic_ostream, 1075–1078
  basic_string, 692
  basic_string_view, 703
  bitset, 538, 539
  byte, 437
  complex, 947
  error_code, 483
  path, 1128
  shared_ptr, 566
  unique_ptr, 557

operator<<
  basic_ostream, 1075–1078
  basic_string, 692
  basic_string_view, 703
  bitset, 538, 539
  byte, 437
  complex, 947
  error_code, 483
  path, 1128
  shared_ptr, 571
  sub_match, 1181
  thread::id, 1218
  time_point, 651
  tuple, 504
  type_index, 654
  unique_ptr, 560, 561
  valarray, 997
  variant, 527
  weak_ordering, 466

operator<<=
  bitset, 536
  byte, 437
  gslice_array, 1002
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>indirect_array</td>
<td>1004</td>
</tr>
<tr>
<td>mask_array</td>
<td>1003</td>
</tr>
<tr>
<td>slice_array</td>
<td>999</td>
</tr>
<tr>
<td>valarray</td>
<td>994, 995</td>
</tr>
<tr>
<td>operator&lt;=</td>
<td>415, 1304</td>
</tr>
<tr>
<td>basic_string</td>
<td>691</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>703</td>
</tr>
<tr>
<td>directory_entry</td>
<td>1138</td>
</tr>
<tr>
<td>duration</td>
<td>647</td>
</tr>
<tr>
<td>monostate</td>
<td>528</td>
</tr>
<tr>
<td>move_iterator</td>
<td>876</td>
</tr>
<tr>
<td>optional</td>
<td>515–517</td>
</tr>
<tr>
<td>pair</td>
<td>494</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>465</td>
</tr>
<tr>
<td>path</td>
<td>1128</td>
</tr>
<tr>
<td>queue</td>
<td>850</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>869</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>570</td>
</tr>
<tr>
<td>stack</td>
<td>854</td>
</tr>
<tr>
<td>strong_ordering</td>
<td>468</td>
</tr>
<tr>
<td>sub_match</td>
<td>1177–1181</td>
</tr>
<tr>
<td>thread:::id</td>
<td>1218</td>
</tr>
<tr>
<td>time_point</td>
<td>651</td>
</tr>
<tr>
<td>tuple</td>
<td>505</td>
</tr>
<tr>
<td>type_index</td>
<td>654</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>560, 561</td>
</tr>
<tr>
<td>valarray</td>
<td>997</td>
</tr>
<tr>
<td>variant</td>
<td>527</td>
</tr>
<tr>
<td>weak_ordering</td>
<td>466</td>
</tr>
<tr>
<td>operator=</td>
<td></td>
</tr>
<tr>
<td>any</td>
<td>531</td>
</tr>
<tr>
<td>atomic</td>
<td>1205</td>
</tr>
<tr>
<td>atomic&lt;floating-point&gt;</td>
<td>1205</td>
</tr>
<tr>
<td>atomic&lt;integral&gt;</td>
<td>1205</td>
</tr>
<tr>
<td>atomic&lt;shared_ptr&lt;T&gt;&gt;</td>
<td>576</td>
</tr>
<tr>
<td>atomic&lt;T&gt;</td>
<td>1205</td>
</tr>
<tr>
<td>atomic&lt;weak_ptr&lt;T&gt;&gt;</td>
<td>578</td>
</tr>
<tr>
<td>back_insert_iterator</td>
<td>870</td>
</tr>
<tr>
<td>bad_alloc</td>
<td>453</td>
</tr>
<tr>
<td>bad_cast</td>
<td>457</td>
</tr>
<tr>
<td>bad_exception</td>
<td>458</td>
</tr>
<tr>
<td>bad_typeid</td>
<td>457</td>
</tr>
<tr>
<td>basic_filebuf</td>
<td>1094</td>
</tr>
<tr>
<td>basic_fstream</td>
<td>1103</td>
</tr>
<tr>
<td>basic_ifstream</td>
<td>1099</td>
</tr>
<tr>
<td>basic_iostream</td>
<td>1071</td>
</tr>
<tr>
<td>basic_iostream</td>
<td>1063</td>
</tr>
<tr>
<td>basic_ostream</td>
<td>1088</td>
</tr>
<tr>
<td>basic_ofstream</td>
<td>1101</td>
</tr>
<tr>
<td>basic_ostream</td>
<td>1073</td>
</tr>
<tr>
<td>basic_ostringstream</td>
<td>1089</td>
</tr>
<tr>
<td>basic_osyncstream</td>
<td>1108</td>
</tr>
<tr>
<td>basic_regex</td>
<td>1175</td>
</tr>
<tr>
<td>basic_streambuf</td>
<td>1056</td>
</tr>
<tr>
<td>basic_string</td>
<td>676</td>
</tr>
<tr>
<td>basic_stringbuf</td>
<td>1085</td>
</tr>
<tr>
<td>basic_stringstream</td>
<td>1091</td>
</tr>
<tr>
<td>basic.syncbuf</td>
<td>1105</td>
</tr>
<tr>
<td>directory_iterator</td>
<td>1139</td>
</tr>
<tr>
<td>enable_shared_from_this</td>
<td>574</td>
</tr>
<tr>
<td>error_code</td>
<td>482</td>
</tr>
<tr>
<td>error_condition</td>
<td>484</td>
</tr>
<tr>
<td>exception</td>
<td>458</td>
</tr>
<tr>
<td>front_insert_iterator</td>
<td>871</td>
</tr>
<tr>
<td>function</td>
<td>610</td>
</tr>
<tr>
<td>future</td>
<td>1251</td>
</tr>
<tr>
<td>gsllice_array</td>
<td>1001, 1002</td>
</tr>
<tr>
<td>indirect_array</td>
<td>1004</td>
</tr>
<tr>
<td>insert_iterator</td>
<td>872</td>
</tr>
<tr>
<td>locale</td>
<td>714</td>
</tr>
<tr>
<td>mask_array</td>
<td>1003</td>
</tr>
<tr>
<td>match_results</td>
<td>1183</td>
</tr>
<tr>
<td>move_iterator</td>
<td>874</td>
</tr>
<tr>
<td>optional</td>
<td>510–512</td>
</tr>
<tr>
<td>ostream_iterator</td>
<td>879</td>
</tr>
<tr>
<td>ostreambuf_iterator</td>
<td>882</td>
</tr>
<tr>
<td>packaged_task</td>
<td>1257</td>
</tr>
<tr>
<td>pair</td>
<td>493, 494</td>
</tr>
<tr>
<td>path</td>
<td>1121</td>
</tr>
<tr>
<td>promise</td>
<td>1249</td>
</tr>
<tr>
<td>raw_storage_iterator</td>
<td>1318</td>
</tr>
<tr>
<td>recursive_directory_iterator</td>
<td>1141</td>
</tr>
<tr>
<td>reference_wrapper</td>
<td>507</td>
</tr>
<tr>
<td>reverse_iterator</td>
<td>867</td>
</tr>
<tr>
<td>shared_future</td>
<td>1253, 1254</td>
</tr>
<tr>
<td>shared_lock</td>
<td>1236</td>
</tr>
<tr>
<td>shared_ptr</td>
<td>565</td>
</tr>
<tr>
<td>slice_array</td>
<td>999</td>
</tr>
<tr>
<td>thread</td>
<td>1219</td>
</tr>
<tr>
<td>tuple</td>
<td>500, 501</td>
</tr>
<tr>
<td>unique_lock</td>
<td>1232</td>
</tr>
<tr>
<td>unique_ptr</td>
<td>556, 557, 559</td>
</tr>
<tr>
<td>valarray</td>
<td>992</td>
</tr>
<tr>
<td>variant</td>
<td>522, 523</td>
</tr>
<tr>
<td>weak_ptr</td>
<td>572, 573</td>
</tr>
<tr>
<td>operator==</td>
<td></td>
</tr>
<tr>
<td>allocator</td>
<td>550</td>
</tr>
<tr>
<td>basic_string</td>
<td>690</td>
</tr>
<tr>
<td>basic_string_view</td>
<td>702</td>
</tr>
<tr>
<td>bitset</td>
<td>538</td>
</tr>
<tr>
<td>complex</td>
<td>947</td>
</tr>
<tr>
<td>directory_entry</td>
<td>1137</td>
</tr>
<tr>
<td>duration</td>
<td>647</td>
</tr>
<tr>
<td>error_category</td>
<td>480</td>
</tr>
<tr>
<td>error_code</td>
<td>484, 485</td>
</tr>
<tr>
<td>error_condition</td>
<td>484, 485</td>
</tr>
<tr>
<td>function</td>
<td>611</td>
</tr>
<tr>
<td>istringstream</td>
<td>878</td>
</tr>
<tr>
<td>istream_iterator</td>
<td>881</td>
</tr>
<tr>
<td>locale</td>
<td>714</td>
</tr>
<tr>
<td>match_results</td>
<td>1185</td>
</tr>
<tr>
<td>memory_resource</td>
<td>580</td>
</tr>
<tr>
<td>monostate</td>
<td>528</td>
</tr>
<tr>
<td>move_iterator</td>
<td>875</td>
</tr>
<tr>
<td>optional</td>
<td>515, 516</td>
</tr>
<tr>
<td>pair</td>
<td>494</td>
</tr>
<tr>
<td>partial_ordering</td>
<td>465</td>
</tr>
<tr>
<td>path</td>
<td>1128</td>
</tr>
</tbody>
</table>
polymorphic_allocator, 583
queue, 850
regex_iterator, 1191
regex_token_iterator, 1192, 1194
reverse_iterator, 868
scoped_allocator_adaptor, 593
shared_ptr, 569
stack, 854
strong_equality, 464
strong_ordering, 468
sub_match, 1177–1180
thread::id, 1218
time_point, 651
tuple, 505
type_index, 654
unique_ptr, 560, 561
valarray, 997
variant, 526
weak_equality, 463
weak_ordering, 466
operator>, 415, 1304
  basic_string, 690, 691
  basic_string_view, 703
directory_entry, 1138
duration, 647
monostate, 528
move_iterator, 876
optional, 515–517
pair, 494
partial_ordering, 465
path, 1128
queue, 850
reverse_iterator, 868
shared_ptr, 570
stack, 854
strong_ordering, 468
sub_match, 1177–1181
thread::id, 1218
time_point, 651
tuple, 505
type_index, 654
unique_ptr, 560, 561
valarray, 997
variant, 527
weak_ordering, 466
operator>>, 415, 1305
  basic_string, 691
  basic_string_view, 703
directory_entry, 1138
duration, 647
monostate, 528
move_iterator, 876
optional, 515–517
pair, 494
partial_ordering, 465
path, 1128
queue, 850
reverse_iterator, 869
shared_ptr, 570
stack, 854
strong_ordering, 468
sub_match, 1177–1181
thread::id, 1218
time_point, 651
tuple, 505
type_index, 654
unique_ptr, 560, 561
valarray, 997
variant, 527
weak_ordering, 466
operator%,
  duration, 647
  valarray, 996
operator%=
  duration, 646
gslice_array, 1002
indirect_array, 1004
mask_array, 1003
slice_array, 999
valarray, 994, 995
operator[]
  basic_string, 678
  basic_string_view, 698
  bitset, 538
  indirect_array, 1003
  map, 815
  mask_array, 1002
  match_results, 1184
  move_iterator, 875
  reverse_iterator, 868
  shared_ptr, 566
  unique_ptr, 559
  unordered_map, 833
  valarray, 992–994
operator%=
  duration, 646
gslice_array, 1002
indirect_array, 1004
mask_array, 1003
slice_array, 999
valarray, 994, 995
operator&
  bitset, 538
  byte, 438
  valarray, 996
operator&=
  atomic<integral>, 1208
  bitset, 536
byte, 438
gslice_array, 1002
indirect_array, 1004
mask_array, 1003
slice_array, 999
valarray, 994, 995
operator&&
  valarray, 997
operator-
  bitset, 538
  byte, 438
  valarray, 996
operator~=
  atomic<integral>, 1208
  bitset, 536
  byte, 438
gslice_array, 1002
indirect_array, 1004
mask_array, 1003
slice_array, 999
valarray, 994, 995
operator~
  bitset, 537
  byte, 438
  valarray, 994
operator|
  bitset, 538
  byte, 437
  valarray, 996
operator||
  atomic<integral>, 1208
  bitset, 536
  byte, 437
gslice_array, 1002
indirect_array, 1004
mask_array, 1003
slice_array, 999
valarray, 994, 995
operator~|
  valarray, 997
optional, 507
  constructor, 508, 509
destructor, 510
emplace, 512, 513
has_value, 514
operator bool, 514
operator!=, 515, 516
operator*, 514
operator->, 513
operator<, 515–517
operator<=, 515–517
operator=, 510–512
operator==, 515, 516
operator>, 515–517
operator>=, 516, 517
reset, 514
swap, 513
value, 514
value_or, 514

value_type, 507
options
  recursive_directory_iterator, 1141
  synchronized_pool_resource, 586
  unsynchronized_pool_resource, 586
ostream, 1033, 1060
ostream_iterator, 878
  constructor, 879
destructor, 879
operator*, 879
operator++, 879
operator=, 879
ostreambuf_iterator, 881
  constructor, 882
  failed, 882
  operator*, 882
  operator++, 882
  operator=, 882
ostreamstream, 1033, 1082
osstrstream, 1310
  constructor, 1310, 1311
freeze, 1311
pcount, 1311
rdbuf, 1311
str, 1311
osyncstream, 1033, 1104
out
  codecvt, 722
out_of_range, 472, 474, 535, 537, 538, 669
  constructor, 474
outer_allocor
  scoped_allocator_adaptor, 591
outer_allocator_type
  scoped_allocator_adaptor, 589
output_iterator_tag, 864
overflow
  basic_filebuf, 1096
  basic_streambuf, 1059
  basic_stringbuf, 1086
strstreambuf, 1307
overflow_error, 472, 475, 535, 537
  constructor, 475
owner_before
  shared_ptr, 566
  weak_ptr, 573
owner_less
  first_argument_type, 1313
  operator(), 574
result_type, 1313
second_argument_type, 1313
owns_lock
  shared_lock, 1237
  unique_lock, 1234

P
  bernoulli_distribution, 972
  binomial_distribution, 973
geometric_distribution, 974
negative_binomial_distribution, 975
packaged_task, 1256
destructor, 1257
get_future, 1257
make_ready_at_thread_exit, 1258
operator(), 1258
operator=, 1257
reset, 1258
swap, 1257, 1258
valid, 1257

pair, 492, 499–501
creator, 492, 493
get, 495
operator! =, 494
operator<, 494
operator<=, 494
operator==, 493, 494
operator>, 494
operator>=, 494
swap, 494

par, 656
par_unseq, 656
param
seed_seq, 970
parent_path
path, 1125
partial_order, 469
partial_ordering, 464
equivalent, 464
greater, 464
less, 464
operator weak_equality, 465
operator!=, 465
operator<, 465
operator<=, 465
operator==, 465
operator>, 465
operator>=, 465
unordered, 464

partial_sort, 922
partial_sort_copy, 923
partial_sum, 1011
partition, 927
partition_copy, 927
partition_point, 928
path, 1114
append, 1122
assign, 1121
begin, 1127
c_str, 1123
clear, 1122
compare, 1124, 1125
concat, 1122
creator, 1120
directory_entry, 1136
empty, 1126
end, 1128
extension, 1125
filename, 1125
generic_string, 1124
generic_u16string, 1124
generic_u32string, 1124
generic_u8string, 1124
generic_wstring, 1124
has_extension, 1126
has_filename, 1126
has_parent_path, 1126
has_relative_path, 1126
has_root_directory, 1126
has_root_name, 1126
has_root_path, 1126
has_stem, 1126
is_absolute, 1126
is_relative, 1126
iterator, 1127
lexically_normal, 1126
lexically_proximate, 1127
lexically_relative, 1127
make_preferred, 1122
native, 1123
operator string_type, 1123
operator! =, 1128
operator==, 1122
operator/, 1128
operator/=, 1121, 1122
operator<, 1128
operator<<, 1128
operator<=, 1128
operator==, 1121
operator>>, 1129
parent_path, 1125
preferred_separator, 1117
relative_path, 1125
remove_filename, 1123
replace_extension, 1123
replace_filename, 1123
root_directory, 1125
root_name, 1125
root_path, 1125
stem, 1125
string, 1124
swap, 1123
u16string, 1124
u32string, 1124
u8string, 1124
value_type, 1117
wstring, 1124
path1
filesystem_error, 1130
path2
filesystem_error, 1130
pbackfail
basic_filebuf, 1096
basic_streambuf, 1058
basic_stringbuf, 1085
strstreambuf, 1308
pbase
basic_stringbuf, 1056
pbump
basic_stringbuf, 1057
pcount
ostreambuf, 1311
streambuf, 1312
streambuf, 1307
peek
basic_istream, 1069
perm_options, 1131
permissions, 1150
 file_status, 1133, 1134
perms, 1131
 perror, 1154
piecewise_constant_distribution, 984
 constructor, 985
densities, 985
intervals, 985
piecewise_construct, 495
piecewise_construct_t, 495
piecewise_linear_distribution, 985, 986
 constructor, 986, 987
densities, 987
intervals, 987
placeholders, 607
plus, 598
 first_argument_type, 1313
 operator(), 598
 result_type, 1313
 second_argument_type, 1313
plus<> , 598
 operator(), 598
pointer
 allocator_traits, 548
 basic_string, 669
 basic_string_view, 695
 iterator_traits, 863
 scoped_allocator_adaptor, 589
pointer_to
 pointer_traits, 544
pointer_to_binary_function
 zombie, 428
pointer_to_unary_function
 zombie, 428
pointer_traits, 544
difference_type, 544
element_type, 544
pointer_to, 544
rebind, 544
to_address, 545
poisson_distribution, 975
 constructor, 975
 mean, 975
polarg
 complex, 948
polymorphic_allocator, 580
allocate, 581
construct, 582, 583
deallocate, 582
destroy, 583
operator!=, 583
operator==, 583
resource, 583
select_on_container_copy_construction, 583
value_type, 580
pool_options, 584
 largest_required_pool_block, 585
max_blocks_per_chunk, 585
pop
 forward_list, 794
priority_queue, 852
recursive_directory_iterator, 1142
pop_back
 basic_string, 682
pop_heap, 933
pos_type
 char_traits, 662
position
 match_results, 1183
positive_sign
 moneypunct, 744
pow, 950, 1016
 complex, 949
valarray, 997
powf, 1016
powl, 1016
pptr
 basic_stringbuf, 1056
precision
 ios_base, 716, 1042
preferred_separator
 path, 1117
prefix
 match_results, 1184
prev, 865
prev_permutation, 938
PRIuFASTN, 1156
PRIuLEASTN, 1156
PRIuMAX, 1156
PRIuN, 1156
PRIuPTR, 1156
PRIiFASTN, 1156
PRIiLEASTN, 1156
PRIiMAX, 1156
PRIiN, 1156
PRIiPTR, 1156
printf, 1154
PRIoFASTN, 1156
PRIoLEASTN, 1156
PRIoMAX, 1156
PRIoN, 1156
PRIoPTR, 1156
priority_queue, 850
constructor, 851, 852
emplace, 852
swap, 853
PRIuFASTN, 1156
PRIuLEASTN, 1156
PRIuMAX, 1156
PRIuN, 1156
PRIuPTR, 1156
PRIxFASTN, 1156
PRIxLEASTN, 1156
PRIxMAX, 1156
PRIxN, 1156
PRIxPTR, 1156
probabilities
    discrete_distribution, 984
proj
    complex, 948
promise, 1248
        constructor, 1249
destructor, 1249
        get_future, 1249
    operator=, 1249
    set_exception, 1249
    set_exception_at_thread_exit, 1250
    set_value, 1249
    set_value_at_thread_exit, 1250
    swap, 1249, 1250
propagate_on_container_copy_assignment
    allocator_traits, 548
    scoped_allocator_adaptor, 590
propagate_on_container_move_assignment
    allocator, 549
    allocator_traits, 548
    scoped_allocator_adaptor, 590
propagate_on_container_swap
    allocator_traits, 548
    scoped_allocator_adaptor, 590
proximate, 1151
proxy
    istreambuf_iterator, 880
ptr
    from_chars_result, 657
to_chars_result, 656
ptr_fun
    zombie, 428
ptrdiff_t, 434
pubimbuf
    basic_streambuf, 1054
pubseekoff
    basic_streambuf, 1055
pubseekpos
    basic_streambuf, 1055
pubsetbuf
    basic_streambuf, 1055
pubsync
    basic_streambuf, 1055
push
    priority_queue, 852
push_back
    basic_string, 680
deque, 790
push_front
    deque, 790
    forward_list, 794
push_heap, 932
put
    basic_ostream, 1077
    money_put, 742
    num_put, 729
time_put, 739
put_money, 1080
put_time, 1081
putback
    basic_istream, 1069
putc, 1154
putchar, 1154
putenv, 469
puts, 1154
putwc, 705
putwchar, 705
pword
    ios_base, 1044
qsort, 435, 938
queue, 848
swap, 850
quick_exit, 417, 435, 448
quiet_NaN
    numeric_limits, 442
quoted, 1081, 1082
radix
    numeric_limits, 441
raise, 470
rand, 435, 987
discouraged, 987
RAND_MAX, 435
random_access_iterator_tag, 864
random_device, 967
    constructor, 968
    entropy, 968
    operator(), 968
random_shuffle
    zombie, 428
range_error, 472, 474
    constructor, 474, 475
rank, 628
ranlux24, 967
ranlux24_base, 967
ranlux48, 967
ranlux48_base, 967
ratio, 636, 637
ratio_equal, 638

Index of library names 1421
ratio_greater, 638
ratio_greater_equal, 638
ratio_less, 638
ratio_less_equal, 638
raw_storage_iterator, 1318
  base, 1319
  constructor, 1318
  operator*, 1318
  operator++, 1318, 1319
  operator=, 1318
rbegin
  basic_string, 677
  basic_string_view, 698
rbegin(C&), 882
rbegin(initializer_list<E>), 883
rbegin(T (&array)[N]), 883
rdbuf
  basic_fstream, 1103
  basic_ifstream, 1099
  basic_ios, 1047
  basic_istream, 1088
  basic_ofstream, 1101
  basic_ostringstream, 1090
  basic_stringstream, 1091
  istringstream, 1310
  ostringstream, 1311
  stringstream, 1312
  wbuffer_convert, 1327
rdstate
  basic_ios, 1049
read
  basic_istream, 1069
read_symlink, 1151
readsome
  basic_istream, 1069
ready
  match_results, 1183
real, 949
  complex, 946, 948
realloc, 435, 552, 1302
rebind
  pointer_traits, 544
rebind_alloc
  allocator_traits, 548
recursion_pending
  recursive_directory_iterator, 1142
recursive_directory_iterator, 1140
  constructor, 1141
  depth, 1142
  disable_recursion_pending, 1142
  increment, 1142
  operator++, 1142
  operator=, 1141
  options, 1141
  pop, 1142
  recursion_pending, 1142
recursive_mutex, 1223
recursive_timed_mutex, 1225
reduce, 1009
ref
  reference_wrapper, 598
reference
  basic_string, 669
  basic_string_view, 695
  iterator_traits, 863
reference_wrapper, 597
  argument_type, 1315
  constructor, 597
  cref, 598
  first_argument_type, 1315
  get, 598
  operator T&, 598
  operator(), 598
  operator=, 597
  ref, 598
  second_argument_type, 1315
  weak_result_type, 1315
refresh
  directory_entry, 1136
regex, 1160
regex_constants, 1166
  error_type, 1168, 1169
  match_flag_type, 1166
  syntax_option_type, 1166
regex_error, 1169, 1171, 1196
  constructor, 1169
regex_iterator, 1190
  constructor, 1191
  increment, 1191
  operator!=, 1191
  operator*, 1191
  operator++, 1191, 1192
  operator->, 1191
  operator=, 1191
regex_match, 1186, 1187
regex_replace, 1188, 1189
regex_search, 1187, 1188
regex_token_iterator, 1192
  constructor, 1194
  end_of_sequence, 1192
  operator!=, 1194
  operator*, 1194
  operator++, 1195
  operator->, 1195
  operator=, 1192, 1194
regex_traits, 1169
  char_class_type, 1169
  isctype, 1170
  length, 1170
  lookup_classname, 1170
  lookup_collatename, 1170
  transform, 1170
  transform_primary, 1170
  translate, 1170
  translate_nocase, 1170
  value, 1171
register_callback
ios_base, 1044
regular expression traits
   isctype, 1196
   lookup_classname, 1196
   lookup_collatename, 1196
   transform_primary, 1196
reinterpret_pointer_cast
   shared_ptr, 571
rel_ops, 487
relative, 1151
relative_path
   path, 1125
relaxed
   memory_order, 1201
release
   memory_order, 1201
   monotonic_buffer_resource, 588
   shared_lock, 1237
   synchronized_pool_resource, 586
   unique_lock, 1234
   unique_ptr, 557
   unsynchronized_pool_resource, 586
remainder, 1016
remainderf, 1016
remove, 917, 1154
   forward_list, 796
   list, 802
   path, 1151
remove_all, 1152
remove_all_extents, 632
remove_const, 631
remove_copy, 917
remove_copy_if, 917
remove_cv, 631
remove_cvref, 633
remove_extent, 632
remove_filename
   path, 1123
remove_if, 917
   forward_list, 796
remove_pointer, 632
remove_prefix
   basic_string_view, 699
remove_reference, 631
remove_suffix
   basic_string_view, 699
remove_volatile, 631
remquo, 1016
remquof, 1016
remquol, 1016
rename, 1152, 1154
rend
   basic_string, 677
   basic_string_view, 698
rend(C&), 883
rend(initializer_list<E>), 883
rend(T (&array)[N]), 883
rep
system_clock, 652
replace, 915
basic_string, 682–684
replace_copy, 916
replace_copy_if, 916
replace_extension
   path, 1123
replace_filename
   directory_entry, 1136
   path, 1123
replace_if, 915
reserve
   basic_string, 677
   vector, 805
reset
any, 532
bitset, 537
optional, 514
packaged_task, 1258
shared_ptr, 566
unique_ptr, 557, 559, 560
weak_ptr, 573
resetiosflags, 1078
resize
   basic_string, 677
deque, 790
   forward_list, 795
   list, 800
   valarray, 996
   vector, 806
resize_file, 1152
resource
   polymorphic_allocator, 583
result_type
   binary_negate, 1316
   bit_and, 1313
   bit_not, 1313
   bit_or, 1313
   bit_xor, 1313
divides, 1313
equal_to, 1313
function, 1313
greater, 1313
greater_equal, 1313
hash, 1315
less, 1313
less_equal, 1313
logical_and, 1313
logical_not, 1313
logical_or, 1313
map::value_compare, 1316
mem_fn, 1315
minus, 1313
modulo, 1313
multimap::value_compare, 1316
multiplies, 1313
negate, 1313
not_equal_to, 1313
owner_less, 1313
Index of library names
Index of library names
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index of library names</td>
<td>1425</td>
</tr>
</tbody>
</table>
set_unexpected
  zombie, 428
set_union, 930
set_value
  promise, 1249
set_value_at_thread_exit
  promise, 1250
setbase, 1079
setbuf, 1154
  basic_filebuf, 1096
  basic_streambuf, 1057, 1087
  strstreambuf, 1309
setenv, 469
setf
  ios_base, 1042
setfill, 1079
setg
  basic_streambuf, 1056
setiosflags, 1079
setjmp, 428, 470
setlocale, 414, 748
setp
  basic_streambuf, 1057
setprecision, 1079
setState
  basic_ios, 1049
setvbuf, 1154
setw, 1079
sgetc
  basic_streambuf, 1055
sgetn
  basic_streambuf, 1055
share
  future, 1251
shared_from_this
  enable_shared_from_this, 575
shared_future, 1252
  constructor, 1253
  destructor, 1253
get, 1254
operator=, 1253, 1254
valid, 1254
wait, 1254
wait_for, 1254
wait_until, 1254
shared_lock, 1234
  constructor, 1235, 1236
  destructor, 1236
  lock, 1236
mutex, 1237
operator bool, 1237
operator=, 1236
owns_lock, 1237
release, 1237
swap, 1237
try_lock, 1236
try_lock_for, 1237
try_lock_until, 1236
unlock, 1237
shared_mutex, 1227
shared_ptr, 562, 575, 1320
  atomic_compare_exchange_strong, 1322
  atomic_compare_exchange_strong_explicit,
    1322
  atomic_compare_exchange_weak, 1322
  atomic_compare_exchange_weak_explicit, 1322
atomic_exchange, 1322
atomic_exchange_explicit, 1322
atomic_is_lock_free, 1321
atomic_load, 1321
atomic_load_explicit, 1321
atomic_store, 1321
atomic_store_explicit, 1322
const_pointer_cast, 570
constructor, 563–565
destructor, 565
dynamic_pointer_cast, 570
get, 566
get_deleter, 571
operator bool, 566
operator!=, 569
operator*, 566
operator>>, 566
operator<, 569
operator<<, 571
operator<=, 570
operator=, 565
operator==, 569
operator>>, 570
operator>=, 570
operator[], 566
owner_before, 566
reinterpret_pointer_cast, 571
reset, 566
static_pointer_cast, 570
swap, 565, 570
unique, 1320
use_count, 566
shared_timed_mutex, 1228
shift
  valarray, 995
showbase, 1049
showmanyc
  basic_filebuf, 1095
  basic_streambuf, 1057, 1095
showpoint, 1050
showpos, 1050
shuffle
  shuffle_order_engine, 965, 966
  constructor, 966
Index of library names

- sig_atomic_t, 470
- SIG_DFL, 470
- SIG_ERR, 470
- SIG_IGN, 470
- SIGABRT, 470
- SIGFPE, 470
- SIGILL, 470
- SIGINT, 470
- signal, 470
- signaling_NaN
  - numeric_limits, 443
- signbit, 1016
- SIGSEGV, 470
- SIGTERM, 470
- sin, 1016
  - complex, 949
  - valarray, 997
- sinh, 1016
- sinh, 1016
  - complex, 949
  - valarray, 997
- sinh, 1016
- sinh, 1016
- size
  - array, 785, 787
  - basic_string, 677
  - basic_string_view, 698
  - bitset, 538
  - gslice, 1001
  - initializer_list, 462
  - match_results, 1183
  - seed_seq, 969
  - slice, 998
  - valarray, 995
- size(C& c), 883
- size(T (&array)[N]), 883
- size_t, 110, 434, 435, 653, 704–706, 1154
- size_type
  - allocator_traits, 548
  - basic_string, 669
  - basic_string_view, 695
  - scoped_allocator_adaptor, 589
- skipws, 1050
- sleep_for
  - this_thread, 1221
- sleep_until
  - this_thread, 1221
- slice, 998
  - constructor, 998
  - size, 998
  - start, 998
  - stride, 998
- slice_array, 998
  - operator==, 999
  - operator+, 999
  - operator-, 999
  - operator=, 999
  - operator<<, 999
  - operator<<, 999
- operator=, 999
- operator>>=, 999
- operatorX, 999
- operatorX, 999
- operator=, 999
- operator=, 999
- operator=, 999
- value_type, 998
- snextc
  - basic_streambuf, 1055
- snprintf, 1154
- sort, 922
  - forward_list, 797
  - list, 802
- sort_heap, 933
- space, 1152
- sph_bessel, 1030
- sph_bessel, 1030
- sph_bessel, 1030
- sph_legendre, 1031
- sph_legendre, 1031
- sph_legendre, 1031
- sph_neumann, 1031
- sph_neumann, 1031
- sph_neumann, 1031
- sph_neumann, 1031
- splice
  - list, 801
  - splice_after
    - forward_list, 795, 796
  - sprintf, 1154
  - sputbackc
    - basic_streambuf, 1055
  - sputc
    - basic_streambuf, 1056
  - sputn
    - basic_streambuf, 1056
- sqrt, 1016
  - complex, 949
  - valarray, 997
  - sqrtf, 1016
  - sqrtl, 1016
  - srand, 435, 987
  - sscanf, 1154
  - stable_partition, 927
  - stable_sort, 922
  - stack, 853
  - constructor, 854
  - swap, 854
  - start
    - galice, 1001
    - slice, 998
  - starts_with
    - basic_string, 688
    - basic_string_view, 700
  - state
    - fpos, 1045
    - wbuffer_convert, 1327
    - wstring_convert, 1325
  - state_type
    - char_traits, 662
Index of library names

- wbbuffer_convert, 1327
- wstring_convert, 1325
- static_pointer_cast
  - shared_ptr, 570
- status, 1152, 1153
- directory_entry, 1137
- status_known, 1154
- stdev
  - normal_distribution, 979
- stderr, 1154
- stdin, 1154
- stdout, 1154
- steady_clock, 652
- stem
  - path, 1125
- stod, 693
- stof, 693
- stoi, 692, 693
- stol, 692, 693
- stold, 693
- stoll, 692, 693
- store
  - atomic, 1205
  - atomic<floating-point>, 1205
  - atomic<integral>, 1205
  - atomic<shared_ptr<T>>, 576
  - atomic<T>, 1205
  - atomic<weak_ptr<T>>, 578
- stoul, 692, 693
- stoull, 692, 693
- str
  - basic_istringstream, 1088
  - basic_ostringstream, 1090
  - basic_stringbuf, 1085
  - basic_stringstream, 1091
  - istreambuf, 1310
  - match_results, 1183
  - ostringstream, 1311
  - streambuf, 1312
  - streambuf, 1307
  - sub_match, 1177
- strcat, 704
- strchr, 704
- strcmp, 704
- strcoll, 704
- strcpy, 704
- strcspn, 704
- streambuf, 1033, 1052
- streamoff, 1038, 1045
- streamsize, 1038
- strerror, 704
- strftime, 653, 739
- stride
  - gslice, 1001
  - slice, 998
- string
  - operator"sv, 703
- stringbuf, 1033, 1082
- stringstream, 1033, 1082
- strlen, 704, 1306, 1307, 1311
- strncat, 704
- strcmp, 704
- strncmp, 704
- strncpy, 704
- strong_equal, 469
- strong_equality, 463
  - equal, 463
  - equivalent, 463
  - nonequal, 463
  - nonequivalent, 463
  - operator weak_equality, 464
  - operator!=, 464
  - operator==, 464
- strong_order, 468
- strong_ordering, 467
  - equal, 467
  - equivalent, 467
  - greater, 467
  - less, 467
  - operator partial_ordering, 467
  - operator strong_equality, 467
  - operator weak_equality, 467
  - operator weak_ordering, 467
  - operator!=, 468
  - operator<, 468
  - operator<=, 468
  - operator==, 468
  - operator>, 468
  - operator>=, 468
- strpbrk, 704
- strrchr, 704
- strspn, 704
- strstr, 704
- strstream, 1311
  - constructor, 1312
  - destructor, 1312
  - freeze, 1312
  - pcount, 1312
  - rdbuf, 1312
  - str, 1312
  - strstreambuf, 1305
  - constructor, 1306, 1307
  - destructor, 1307
  - freeze, 1307
  - overflow, 1307
  - pbackfail, 1308
  - pcount, 1307
  - seekoff, 1308
  - seekpos, 1309
  - setbuf, 1309
  - str, 1307
  - underflow, 1308
- strtod, 435
- strtof, 435
- strtoimax, 1156
- strtok, 704
strtol, 435
strtold, 435
strtoll, 435
strtoul, 435
strtoull, 435
strtoumax, 1156
strxfrm, 704
student_t_distribution, 982
constructor, 982
mean, 982
sub_match, 1176
compare, 1177
constructor, 1177
length, 1177
operator basic_string, 1177
operator!=, 1177–1180
operator<, 1177–1180
operator<=, 1177–1181
operator==, 1177–1180
operator>, 1177–1180
operator>=, 1177–1181
str, 1177
substr
basic_string, 687
basic_string_view, 699
subtract_with_carry_engine, 962
constructor, 963
suffix
match_results, 1185
sum
valarray, 995
swap
basic_streambuf, 1055
basic_filebuf, 1094
basic_fstream, 1103
basic_ifstream, 1099
basic_ios, 1048
basic_iostream, 1071
basic_istream, 1063
basic_ostreamstream, 1088
basic_ostream, 1101
basic_ostream, 1073
basic_ostreamstream, 1090
basic_regex, 1176
basic_streambuf, 1056
basic_string, 684, 691
basic_string_view, 699
basic_stringbuf, 1085
basic_stringstream, 1091
basic_systream, 1106
deque, 791
forward_list, 797
function, 610, 611
list, 803
map, 817
match_results, 1185
multimap, 820
multiset, 826
optional, 513, 517
packaged_task, 1257, 1258
pair, 494
path, 1123, 1128
priority_queue, 853
promise, 1249, 1250
queue, 850
set, 823
shared_lock, 1237
shared_ptr, 565, 570
stack, 854
thread, 1219, 1220
tuple, 501
unique_lock, 1234
unique_ptr, 557
unordered_map, 834
unordered_multimap, 839
unordered_multiset, 847
unordered_set, 843
valarray, 995, 998
variant, 525, 528
vector, 806, 807
vector<bool>, 809
weak_ptr, 573
swap(unique_ptr&, unique_ptr&), 560
swap_ranges, 914
swprintf, 705
swscanf, 705
eysmlink_status, 1154
directory_entry, 1137
sync
basic_filebuf, 1097
basic_istream, 1069
basic_streambuf, 1057
basic_systream, 1106
sync_with_stdio
ios_base, 1043
syncbuf, 1033, 1104
synchronized_pool_resource, 584
constructor, 586
destructor, 586
do_allocate, 586
do_deallocate, 586
do_is_equal, 586
options, 586
release, 586
upstream_resource, 586
syntax_option_type, 1166
awk, 1166, 1167
basic, 1166, 1167
collate, 1166, 1167, 1196
ECMAScript, 1166, 1167
egrep, 1166, 1167
extended, 1166, 1167
grep, 1166, 1167
icase, 1166, 1167
Index of library names 1429
multiline, 1167
nosubs, 1166, 1167
optimize, 1166, 1167
system, 435, 469
system_category, 479, 481
system_clock, 652
from_time_t, 652
rep, 652
to_time_t, 652
system_error, 477, 485
code, 486
constructor, 485, 486
what, 486
t
    binomial_distribution, 973
table
cctype<char>, 720
tan, 1016
    complex, 949
    valarray, 997
tanf, 1016
tanh, 1016
    complex, 949
    valarray, 997
tanhf, 1016
tanh, 1016
tanl, 1016
target
    function, 611
target_type
    function, 611
tellg
    basic_istream, 1070
tellp
    basic_ostream, 1074
temp_directory_path, 1154
terminate, 448, 459
terminate_handler, 429, 459
test
    bitset, 538
test_and_set
    atomic_flag, 1212
tgamma, 1016
tgammaf, 1016
tgammal, 1016
this_thread
    get_id, 1220
    sleep_for, 1221
    sleep_until, 1221
    yield, 1221
thousands_sep
    moneypunct, 744
    numpunct, 732
thread, 1217
    constructor, 1219
    destructor, 1219
detach, 1220
gid, 1220
hardware_concurrency, 1220
id, 1217
join, 1220
joinable, 1219
operator!=, 1219
swap, 1219, 1220
thread::id, 1217
    constructor, 1218
    operator!=, 1218
    operator<, 1218
    operator<<, 1218
    operator<=, 1218
    operator==, 1218
    operator>, 1218
    operator>=, 1218
throw_with_nested
    nested_exception, 461
tie, 502
    basic_ios, 1047
time, 653
time_get, 735
date_order, 736
do_date_order, 737
do_get, 738
do_get_date, 737
do_get_monthname, 737
do_get_time, 737
do_get_weekday, 737
do_get_year, 738
get, 736
gid, 736
gid::monthname, 736
gid::time, 736
gid::weekday, 736
gid::year, 736
time_get_byname, 738
time_point, 649
ceil, 651
    constructor, 650
floor, 651
max, 650
min, 650
operator!=, 651
operator+, 650
operator==, 650
operator-, 650, 651
operator-=, 650
operator<, 651
operator<=, 651
operator==, 651
operator>, 651
operator>=, 651
round, 652
time_point_cast, 651
time_since_epoch, 650
time_point_cast, 651
time_put, 738
do_put, 739
put, 739
Index of library names

time_putbyname, 739
time_since_epoch
time_point, 650
time_t, 653
TIME_UTC, 653
timespec_mutex, 1225
timespec, 653
timespec_get, 653
tinyness_before
    numeric_limits, 443
tm, 653, 705
TMP_MAX, 1154
tmpfile, 1154
tmpnam, 1154
to_address, 545
to_bytes
    wstring_convert, 1326
to_chars, 657, 658
to_chars_result, 656
ec, 656
ptr, 656
to_integer
    byte, 438
to_string, 693
    bitset, 537
to_time_t
    system_clock, 652
to_ullong
    bitset, 537
to_ulong
    bitset, 537
to_wstring, 694
tolower, 704, 716
case, 717
case<char>, 720
toupper, 704, 715
case, 717
case<char>, 720	owctrans, 704
towlower, 704
towupper, 704
traits_type
    basic_string, 669
    basic_string_view, 695
transform, 914
    collate, 734
    regex_traits, 1170
transform_exclusive_scan, 1013
transform_inclusive_scan, 1014
transform_primary
    regex_traits, 1170
transform_reduce, 1010, 1011
translate
    regex_traits, 1170
translate_nocase
    regex_traits, 1170
traps
    numeric_limits, 443
treat_as_floating_point, 642
true_type, 621
truename
    num punct, 733
trunc, 1016
truncf, 1016
trunc1, 1016
try_emplace
    map, 816
unordered_map, 833, 834
try_lock, 1237
    shared_lock, 1236
    unique_lock, 1233
try_lock_for
    shared_lock, 1237
    unique_lock, 1233
try_lock_until
    shared_lock, 1236
    unique_lock, 1233
try_to_lock, 1229
try_to_lock_t, 1229
tuple, 495, 497, 787
    constructor, 498–500
    forward_as_tuple, 501
    get, 503, 504
    make_tuple, 501
    operator!=, 505
    operator<, 504
    operator<=, 505
    operator=, 500, 501
    operator==, 504
    operator>, 505
    operator>={}, 505
    swap, 501
tie, 502
tuple_cat, 502
tuple_element, 495, 503, 787
tuple_size, 495, 503, 787
    in general, 503
type
    any, 532
    file_status, 1133, 1134
type_index, 653
    constructor, 654
    hash_code, 654
    name, 654
    operator!=, 654
    operator<, 654
    operator<=, 654
    operator=, 654
    operator>, 654
    operator>={}, 654
type_info, 104, 455, 456
    before, 456
    hash_code, 456
    name, 456
    operator!=, 456
    operator==, 456

© ISO/IEC
un16string
   operator"s, 694
   path, 1124
un16string_view
   operator"sv, 703
u32string
   operator"s, 694
   path, 1124
u32string_view
   operator"sv, 703
u8path, 1129
u8string
   path, 1124
UCHAR_MAX, 445
uflow
   basic_filebuf, 1096
   basic_streambuf, 1058
uint16_t, 446
uint32_t, 446
uint64_t, 446
uint8_t, 446
uint_fast16_t, 446
uint_fast32_t, 446
uint_fast64_t, 446
uint_fast8_t, 446
uint_least16_t, 446
uint_least32_t, 446
uint_least64_t, 446
uint_least8_t, 446
UINT_MAX, 445
uintmax_t, 446
uintptr_t, 446
ULLONG_MAX, 445
ULONG_MAX, 445
unary_function
   zombie, 428
unary_negate, 1316
   argument_type, 1316
   operator(), 1316
   result_type, 1316
uncaught_exception, 1312
uncaught_exceptions, 393, 459
undeclare_no_pointers, 546
undeclare_reachable, 545
underflow
   basic_filebuf, 1095
   basic_streambuf, 1058
   basic_stringbuf, 1085
   stringstream, 1308
underflow_error, 472, 475
   constructor, 475
underlying_type, 633
unexpected
   zombie, 428
unexpected_handler
   zombie, 428
unget
   basic_istream, 1069
   ungetc, 1154
ungetwc, 705
uniform_int_distribution, 970
   a, 971
   b, 971
   constructor, 971
uniform_real_distribution, 971
   a, 972
   b, 972
   constructor, 972
uninitialized_copy, 551
uninitialized_copy_n, 551
uninitialized_default_construct, 550
uninitialized_default_construct_n, 550
uninitialized_fill, 551
uninitialized_fill_n, 551
uninitialized_move, 551
uninitialized_move_n, 551
uninitialized_value_construct, 550
uninitialized_value_construct_n, 550
unique, 918
   forward_list, 796
   list, 802
   shared_ptr, 1320
unique_copy, 918
unique_lock, 1230
   constructor, 1231, 1232
   destructor, 1232
   lock, 1233
   mutex, 1234
   operator bool, 1234
   operator=, 1232
   owns_lock, 1234
   release, 1234
   swap, 1234
   try_lock, 1233
   try_lock_for, 1233
   try_lock_until, 1233
   unlock, 1233
unique_ptr, 554, 558, 565
   constructor, 555, 556, 559
   destructor, 556
   get, 557
   get_deleter, 557
   operator bool, 557
   operator!=, 560, 561
   operator*, 557
   operator->, 557
   operator<, 560, 561
   operator<<, 561
   operator<=, 560, 561
   operator>>, 556, 557, 559
   operator>>, 560, 561
   operator>, 560, 561
   operator>>, 560, 561
   operator[](), 559
   release, 557
   reset, 557, 559, 560
   swap, 557
unitbuf, 1050
N4713

© ISO/IEC

unlock
shared_lock, 1237
unique_lock, 1233
unordered
partial_ordering, 464
unordered_map, 827–829
at, 833
constructor, 832, 833
insert, 833
insert_or_assign, 834
operator[], 833
swap, 834
try_emplace, 833, 834
unordered_multimap, 827, 835
constructor, 838
insert, 839
swap, 839
unordered_multiset, 828, 843
constructor, 846
swap, 847
unordered_set, 828, 839
constructor, 842
swap, 843
unsetf
ios_base, 1042
unshift
codecvt, 722
unsynchronized_pool_resource, 584
constructor, 586
destructor, 586
do_allocate, 586
do_deallocate, 586
do_is_equal, 586
options, 586
release, 586
upstream_resource, 586
upper_bound, 925
uppercase, 1050
upstream_resource
monotonic_buffer_resource, 588
synchronized_pool_resource, 586
unsynchronized_pool_resource, 586
use_count
shared_ptr, 566
weak_ptr, 573
use_facet
locale, 715
uses_allocator, 546
promise, 1248
uses_allocator<tuple>, 505
USHRT_MAX, 445
va_arg, 469
va_copy, 469
va_end, 428, 469
va_list, 428, 469
va_start, 469, 470
valarray, 989, 1001
apply, 996
Index of library names

constructor, 991
cshift, 996
destructor, 992
max, 995
min, 995
operator!, 994
operator!=, 997
operator*, 996
operator*=, 994, 995
operator+, 994, 996
operator+=, 994, 995
operator-, 994, 996
operator-=, 994, 995
operator/, 996
operator/=, 994, 995
operator<, 997
operator<<, 996
operator<<=, 994, 995
operator<=, 997
operator=, 992
operator==, 997
operator>, 997
operator>=, 997
operator>>, 996
operator>>=, 994, 995
operator[], 992–994
operator%, 996
operator%=, 994, 995
operator&, 996
operator&=, 994, 995
operator&&, 997
operator^, 996
operator^=, 994, 995
operator , 994
operator|, 996
operator|=, 994, 995
operator||, 997
resize, 996
shift, 995
size, 995
sum, 995
swap, 995, 998
valid
future, 1252
packaged_task, 1257
shared_future, 1254
value
error_code, 482
error_condition, 484
optional, 514
regex_traits, 1171
value_or
optional, 514
value_type
allocator, 549
atomic, 1204
basic_string, 669
basic_string_view, 695
complex, 944

~

1433


gslice_array, 1001
indirect_array, 1003
integer_sequence, 491
integral_constant, 621
iterator_traits, 863
mask_array, 1002
optional, 507
path, 1117
polymorphic_allocator, 580
scoped_allocator_adaptor, 589
slice_array, 998
valueless_by_exception, variant, 524
variant, 519
compiler, 520–522
destructor, 522
emplace, 524
get, 526
get_if, 526
holds_alternative, 526
index, 525
operator!=, 527
operator<, 527
operator<=, 527
operator==, 526
operator>=, 527
operator>, 527
operator>=, 527
swap, 525
valueless_by_exception, 524
visit, 527
variant_alternative, 525, 526
variant_size, 525
vector, 803
constructor, 805
operator<, 805
operator==, 805
swap, 807
vector< bool >, 807
flip, 809
swap, 809
vfprintf, 1154
vscanf, 1154
vfprintf, 705
vfwscanf, 705
visit, 527
void_pointer
allocator_traits, 548
scoped_allocator_adaptor, 589
vprintf, 1154
vscanf, 1154
vsnprintf, 1154
vsprintf, 1154
vsscanf, 705
vfwscanf, 705
visit, 705
vscanf, 705
wait
condition_variable, 1241
condition_variable_any, 1244
future, 1252
shared_future, 1254
wait_for
condition_variable, 1242, 1243
condition_variable_any, 1245
future, 1252
shared_future, 1254
wait_until
condition_variable, 1241, 1242
condition_variable_any, 1244, 1245
future, 1252
shared_future, 1254
wbuffer_convert, 1326
constructor, 1327
destructor, 1327
dbuf, 1327
state, 1327
state_type, 1327
wcerr, 1036
WCHAR_MAX, 705
WCHAR_MIN, 705
wcin, 1036
wclog, 1036
wcout, 1036
wcstombs, 707
wcrtomb, 705, 707
wscat, 705
wscmp, 705
wscoll, 705
wcsncpy, 705
wcschr, 705
wcscat, 1327
wscpy, 705
wcsdup, 705
wcsxfrm, 705
wcstok, 705
wcstok, 705
wcstol, 705
wcstold, 705
wcstoll, 705
wcstoull, 705
wcstoumax, 705
wcsxfrm, 705
wcstoumax, 1156
wcstok, 705
wcstol, 705
wcstold, 705
wcstoll, 705
wcstombs, 435, 707
wcstoul, 705
wcstoull, 705
wcstoumax, 1156
wcstoull, 435, 707
Index of library names

wctrans, 704
wctrans_t, 704
wctype, 704
wctype_t, 704
weak_equal, 469
weak_equality, 463
equivalent, 463
nonequivalent, 463
operator!=, 463
operator==, 463

weak_from_this
enable_shared_from_this, 575
weak_order, 468
weak_ordering, 465
equivalent, 465
greater, 465
less, 465
operator partial_ordering, 466
operator weak_equality, 466
operator!=, 466
operator<, 466
operator<=, 466
operator==, 466
operator>, 466
operator>=-, 466

weak_ptr, 565, 571, 575
constructor, 572
destructor, 572
expired, 573
lock, 573
operator=-, 572, 573
owner_before, 573
reset, 573
swap, 573
use_count, 573
weakly_canonical, 1154
weibull_distribution, 977
a, 977
b, 977
constructor, 977

WEOF, 704, 705
wfilebuf, 1033, 1091
wfstream, 1033, 1091

what
bad_alloc, 454
bad_any_cast, 529
bad_array_new_length, 454
bad_cast, 457
bad_exception, 459
bad_function_call, 608
bad_optional_access, 515
bad_typeid, 457
bad_variant_access, 528
bad_weak_ptr, 562
exception, 458
filesystem_error, 1130
future_error, 1247
system_error, 486

wide_string
wstring_convert, 1326
widen
basic_ios, 1047
ctype, 717
ctype<char>, 720
width
ios_base, 716, 1042
wifstream, 1033, 1091
wint_t, 704, 705
wios, 1037
wiostream, 1033, 1060
wistringstream, 1033, 1082
wmemchr, 705
wmemcmp, 705
wmemcpy, 705
wmemmove, 705
wmemset, 705
wostream, 1033, 1091
wofstream, 1033, 1060
wstringstream, 1033, 1082
wofstream, 1033, 1091
wostream, 1033, 1060
wstringstream, 1033, 1082
wsyncstream, 1033, 1104
wprintf, 705
wregex, 1160
write
basic_ostream, 1077
ws, 1065, 1070
wscanf, 705
wstreambuf, 1033, 1052
wstring
operator"s, 694
path, 1124
wstring_convert, 1324
byte_string, 1325
constructor, 1326
destructor, 1326
converted, 1325
to_bytes, 1325
int_type, 1325
state, 1325
to_bytes, 1326
state_type, 1325

wide_string, 1326
wstringstream
operator"sv, 703
wstringstream, 1033, 1082
wstringstream, 1033, 1082
wstringstream, 1033, 1104
xalloc
ios_base, 1043
xsgetn
basic_streambuf, 1058
xsputn
basic_streambuf, 1059

yield
this_thread, 1221

zero
duration, 646
duration_values, 643
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, 404
additional execution policies supported by parallel algorithms, 655, 904
additional file_type enumerators for file systems supporting additional types of file, 1131
additional formats for time_get::do_get_date, 737
additional supported forms of preprocessing directive, 396
algorithms for producing the standard random number distributions, 970
alignment, 57
alignment additional values, 57
alignment of bit-fields within a class object, 222
allocation of bit-fields within a class object, 222
any use of an invalid pointer other than to perform indirection or deallocate, 53
argument values to construct basic_ios::failure, 1049
assignability of placeholder objects, 607
behavior of iostream classes when traits::pos_type is not streampos or when traits::off_type is not streamoff, 1033
behavior of non-standard attributes, 174
bits in a byte, 48
choice of larger or smaller value of floating literal, 19
concatenation of some types of string literals, 21
conversions between pointers and integers, 107
converting characters from source character set to execution character set, 9
converting function pointer to object pointer and vice versa, 107
default configuration of a pool, 586
default next_buffer_size for a monotonic_buffer_resource, 587
default number of buckets in unordered_map, 833
default number of buckets in unordered_multimap, 838
default number of buckets in unordered_multiset, 846, 847
default number of buckets in unordered_set, 842, 843
defining main in freestanding environment, 70
definition and meaning of __STDC_VERSION__, 405
definition of NULL, 436, 1302
derived type for typeid, 104
diagnostic message, 3
dynamic initialization of static inline variables before main, 73
dynamic initialization of static variables before main, 72, 73
dynamic initialization of thread-local variables before entry, 73
effect of calling associated Laguerre polynomials with n >= 128 or m >= 128, 1026
effect of calling associated Legendre polynomials with l >= 128, 1026
effect of calling basic_filebuf::setbuf with nonzero arguments, 1096
effect of calling basic_filebuf::sync when a get area exists, 1097
effect of calling basic_streambuf::setbuf with nonzero arguments, 1087
effect of calling cylindrical Bessel functions of the first kind with nu >= 128, 1028
effect of calling cylindrical Neumann functions with nu >= 128, 1028
effect of calling Hermite polynomials with n >= 128, 1029
effect of calling ios_base::sync_with_stdio after any input or output operation on standard streams, 1043
effect of calling irregular modified cylindrical Bessel functions with nu >= 128, 1028
effect of calling Laguerre polynomials with n >= 128, 1030
effect of calling Legendre polynomials with l >= 128, 1030
effect of calling regular modified cylindrical Bessel functions with nu >= 128, 1027
effect of calling spherical associated Legendre functions with l >= 128, 1031
effect of calling spherical Bessel functions with n >= 128, 1030
effect of calling spherical Neumann functions with n >= 128, 1031
effect of filesystem::copy, 1143
effect on C locale of calling locale::global, 715
encoding of universal character name not in execution character set, 18
 error_category for errors originating outside the operating system, 433
exception type when `random_device` constructor fails, 968
exception type when `random_device::operator()` fails, 968
exception type when `shared_ptr` constructor fails, 564
exceptions thrown by standard library functions that have a potentially-throwing exception specification, 433
execution character set and execution wide-character set, 10
exit status, 448
extended signed integer types, 60
file type of the file argument of `filesystem::status`, 1153
formatted character sequence generated by `time_put::do_put` in C locale, 739
forward progress guarantees for implicit threads of parallel algorithms (if not defined for `thread`), 903
growth factor for `monotonic_buffer_resource`, 588
headers for freestanding implementation, 417
how `random_device::operator()` generates values, 968
interactive device, 7
interpretation of the path character sequence with format `path::auto_format`, 1131
largest supported value to configure the largest allocation satisfied directly by a pool, 585
largest supported value to configure the maximum number of blocks to replenish a pool, 585
linkage of `main`, 71
linkage of names from C standard library, 418
linkage of objects between C++ and other languages, 173
locale names, 713
lvalue-to-rvalue conversion of an invalid pointer value, 76
manner of search for included source file, 398
mapping from name to catalog when calling `messages::do_open`, 745
mapping from physical source file characters to basic source character set, 10, 1296
mapping header name to header or external source file, 13
mapping of pointer to integer, 106
mapping physical source file characters to basic source character set, 9
mapping to message when calling `messages::do_get`, 746
maximum depth of recursive template instantiations, 359
maximum size of an allocated object, 112, 454
meaning of ‘’, ‘\’, ‘/’, ‘*’, or ‘/’ in a q-char-sequence or an h-char-sequence, 13
meaning of `asm` declaration, 171
meaning of attribute declaration, 140
meaning of `dot-dot` in `root-directory`, 1118
negative value of character literal in preprocessor, 397
nesting limit for `#include` directives, 398
NTCTS in `basic_ostream<charT, traits>& operator<<(nullptr_t)`, 1076
number of placeholders for bind expressions, 595, 607
number of threads in a program under a freestanding implementation, 66
numeric values of character literals in `#if` directives, 397
operating system on which implementation depends, 1109
parameters to `main`, 71
passing argument of class type through ellipsis, 101
physical source file characters, 9
presence and meaning of `native_handle_type` and `native_handle`, 1214
range defined for character literals, 18
rank of extended signed integer type, 63
representation of `char`, 59
required libraries for freestanding implementation, 6
resource limits on a message catalog, 746
result of `filesystem::file_size`, 1148
result of inexact floating-point conversion, 78
result of right shift of negative value, 119
return value of `bad_alloc::what`, 454
return value of `bad_any_cast::what`, 529
return value of `bad_array_new_length::what`, 454
return value of `bad_cast::what`, 457
return value of `bad_exception::what`, 459
return value of `bad_function_call::what`, 608
return value of `bad_optional_access::what`, 515
return value of `bad_typeid::what`, 457
return value of `bad_variant_access::what`, 528
return value of `char_traits<char16_t>::eof`, 664
return value of `char_traits<char32_t>::eof`, 664
return value of `exception::what`, 458
return value of `type_info::name()`, 456
search locations for “” header, 398
search locations for `<>` header, 398
semantics and default value of token parameter to random_device constructor, 968
semantics of an access through a volatile glvalue, 149
semantics of linkage specification on templates, 307
semantics of linkage specifiers, 171
semantics of non-standard escape sequences, 18
semantics of parallel algorithms invoked with implementation-defined execution policies, 904
sequence of places searched for a header, 398
set of character types that iostreams templates can be instantiated for, 712, 1033
signedness of char, 59, 150
sizeof applied to fundamental types other than char, signed char, and unsigned char, 110
stack unwinding before call to std::terminate(), 390, 393
start-up and termination in freestanding environment, 70
string resulting from __func__, 193
support for extended alignment, 634
support for extended alignments, 57
support for over-aligned types, 1317, 1319
supported multibyte character encoding rules, 663, 665
supported root-names in addition to any operating system dependent root-names, 1117, 1118
text of __DATE__ when date of translation is not available, 404
text of __TIME__ when time of translation is not available, 405
threads and program points at which deferred dynamic initialization is performed, 72, 73
type of a directory-like file, 1138, 1140
type of array::const_iterator, 786
type of array::iterator, 786
type of basic_string::const_iterator, 669
type of basic_string::iterator, 669
type of basic_string_view::const_iterator, 696, 698
type of default_random_engine, 967
type of deque::const_iterator, 788
type of deque::iterator, 788
type of filesystem trivial clock, 1114
type of forward_list::const_iterator, 792
type of forward_list::iterator, 792
type of list::const_iterator, 798
type of list::iterator, 798
type of map::const_iterator, 812
type of map::iterator, 812
type of multimap::const_iterator, 817
type of multimap::iterator, 817
type of multiset::const_iterator, 824
type of multiset::iterator, 824
type of ptdiff_t, 119, 437
type of regex_constants::error_type, 1168
type of regex_constants::match_flag_type, 1167
type of set::const_iterator, 821
type of set::iterator, 821
type of size_t, 437
type of streamoff, 663
type of streampos, 663
type of syntax_option_type, 1166
type of ui16streampos, 664
type of ui32streampos, 664
type of unordered_map::const_iterator, 829
type of unordered_map::const_local_iterator, 829
type of unordered_map::iterator, 829
type of unordered_multimap::const_iterator, 835
type of unordered_multimap::const_local_iterator, 835
type of unordered_multimap::iterator, 835
type of unordered_multiset::const_iterator, 843
type of unordered_multiset::const_local_iterator, 843
type of unordered_multiset::iterator, 843
type of unordered_multiset::local_iterator, 843
type of unordered_set::const_iterator, 839
type of unordered_set::const_local_iterator, 839
type of unordered_set::iterator, 839
type of unordered_set::local_iterator, 839
type of vector<bool>::const_iterator, 807
type of vector<bool>::iterator, 807
type of wstreampos, 665
underlying type for enumeration, 157
value of bit-field that cannot represent assigned value, 125
incremented value, 103
initializer, 199
value of character literal outside range of corresponding type, 18
value of ctype<char>::table_size, 720
value of multicharacter literal, 17
value of pow(0,0), 949

Index of implementation-defined behavior 1439
value of result of inexact integer to floating-point
conversion, 78
value of result of unsigned to signed conversion, 78
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, 60
value representation of pointer types, 61
values of a trivially copyable type, 58
values of various ATOMIC_..._LOCK_FREE macros, 1203

whether `<cfenv>` functions can be used to manage
floating-point status, 942
whether a given atomic type’s operations are
always lock free, 575, 577, 1204, 1205, 1207, 1208, 1210
whether an implementation has relaxed or strict
pointer safety, 56
whether functions from Annex K of the C
standard library are declared when C++
headers are included, 417
whether `get_pointer_safety` returns
pointer_safety::relaxed or
pointer_safety::preferred if the
implementation has relaxed pointer
safety, 546
whether locale object is global or per-thread, 711
whether pragma `FENV_ACCESS` is supported, 942
whether `rand` may introduce a data race, 987
whether sequence pointers are copied by
`basic_filebuf` move constructor, 1093
whether sequence pointers are copied by
`basic_stringbuf` move constructor, 1084
whether source of translation units must be
available to locate template definitions, 10
whether stack is unwound before calling
`std::terminate()` when a noexcept
specification is violated, 393
whether the lifetime of a parameter ends when the
callee returns or at the end of the
enclosing full-expression, 100
whether the thread that executes `main` and the
threads created by `std::thread` provide
concurrent forward progress guarantees, 70
whether `time_get::do_get_year` accepts
two-digit year numbers, 738
whether values are rounded or truncated to the
required precision when converting
between `time_t` values and `time_point`
objects, 652
whether `variant` supports over-aligned types, 520
which functions in the C++ standard library may
be recursively reentered, 432

Index of implementation-defined behavior 1440